Escuela Politécnica Superior

veelu

App móvil para planificar viajes en grupo

Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autora:
Esther Navarro García

Tutor:
José Vicente Berná Martínez

Universitat d'Alacant
Universidad de Alicante

Julio 2020

Resumen

Viajar ha dejado de ser un privilegio que solo unos pocos podían permitirse. Hoy en día, son muchas las personas que viajan para explorar lugares desconocidos o escapar del día a día. Además, cada vez es más común la gente que planifica viajes desde cero, sin delegar esta tarea a agencias especializadas, lo cual reduce significativamente los costes.

Sin embargo, a pesar de ser una alternativa más barata, organizar un viaje resulta un proceso tedioso y que requiere un gran esfuerzo para no tener problemas en el futuro. Esta tarea se vuelve mucho más difícil si se trata de un viaje en grupo, en el que los participantes tienen que ponerse de acuerdo y trabajar de forma colaborativa.

Lo más probable es que estas personas se organicen y recojan la información necesaria usando aplicaciones como *WhatsApp*, *Google Docs* o las notas del móvil. El problema con estas *apps* es que son genéricas, es decir, no están diseñadas con el propósito de planear viajes. Esto supone emplear tiempo extra para ordenar y clasificar toda la información.

Si se busca en la *Play Store*, se pueden encontrar aplicaciones centradas en agilizar aspectos concretos de un viaje. Por ejemplo, existen *apps* para controlar lo que se va gastando, anotar qué pertenencias llevar en la maleta, etc. A pesar de que cumplen su función, sigue siendo un inconveniente tener que descargar y utilizar 3 o 4 aplicaciones para planificar un viaje, pues los datos acaban dispersos e incluso pueden perderse.

En definitiva, resulta necesaria una *app* todo en uno que ayude a gestionar los distintos aspectos de un viaje y que, además, permita hacerlo de forma colaborativa. Por estos motivos se ha desarrollado *veelu*, una aplicación móvil para organizar viajes en grupo.

Gracias a *veelu*, los usuarios pueden planear las actividades que van a realizar durante un viaje, anotar en un *checklist* qué llevar en la maleta y tener un control de los gastos que van teniendo. Además de estas herramientas, *veelu* también permite a los participantes de un viaje escribir sus pensamientos y experiencias mediante entradas de diario, así como compartir fotografías, documentos y otros archivos multimedia.

Abstract

Traveling is no longer a privilege that only a few could afford. Today, many people travel to explore unknown places or escape from the day to day routine. In addition, it is increasingly common for people to plan trips from scratch, without delegating this task to specialized agencies, which significantly reduces costs.

However, despite being a cheaper alternative, organizing a trip is a tedious process that requires a lot of effort to avoid future problems. This task becomes much more difficult if it is a group trip, in which the participants have to agree and work collaboratively.

Most likely, these people organize and collect the necessary information using applications such as *WhatsApp*, *Google Docs* or mobile notes. The problem with these apps is that they are generic, that is, they are not designed for the purpose of planning trips. This involves spending extra time to sort and classify all the information.

If you search the *Play Store*, you can find applications focused on speeding up specific aspects of a trip. For example, there are apps to control what is being spent, write down what belongings to carry in the suitcase, etc. Although they fulfill their function, it is still a drawback having to download and use 3 or 4 applications to plan a trip, since the data ends up scattered and may even be lost.

In short, an all-in-one application is needed to help manage the different aspects of a trip and, also, to do it collaboratively. For these reasons *veelu*, a mobile application for organizing group trips, has been developed.

Thanks to *veelu*, users can plan the activities they are going to do during a trip, write down in a checklist what to carry in their suitcase and keep track of their expenses. In addition to these tools, *veelu* also allows trip participants to write their thoughts and experiences through journal entries, as well as share photos, documents and other multimedia files.

Resum

Viatjar ha deixat de ser un privilegi que només uns pocs podien permetre's. Hui en dia, són moltes les persones que viatgen per explorar llocs desconeguts o escapar del dia a dia. A més, cada vegada és més comú que la gent planifique viatges des de zero, sense delegar aquesta tasca en agències especialitzades, la qual cosa redueix significativament els costos.

No obstant això, tot i ser una alternativa més barata, el fet d'organitzar un viatge resulta un procés tediós i que requereix un gran esforç per no tindre problemes en el futur. Aquesta tasca es torna molt més difícil si es tracta d'un viatge en grup, en el qual els participants han de posar-se d'acord i treballar de forma col·laborativa.

Allò més probable és que aquestes persones s'organitzen i arrepleguen la informació necessària mitjançant l'ús d'aplicacions com *WhatsApp, Google Docs* o les notes del mòbil. El problema amb aquestes *apps* és que són genèriques, és a dir, no estan dissenyades amb el propòsit de planejar viatges. Açò suposa emprar temps extra per ordenar i classificar tota la informació.

Si es fa una cerca en la Play Store, es poden trobar aplicacions centrades a agilitzar aspectes concrets d'un viatge. Per exemple, existeixen *apps* per a controlar el que es va gastant, anotar quines pertinences portar en la maleta, etc. Tot i que compleixen la seua funció, continua sent un inconvenient haver de descarregar i utilitzar 3 o 4 aplicacions per a planificar un viatge, perquè les dades acaben disperses i, fins i tot, poden perdre's.

Definitivament, resulta necessària una aplicació tot en un que ajude a gestionar els distints aspectes d'un viatge i que, a més, permeta fer-ho de forma col·laborativa. Per aquests motius s'ha desenvolupat *veelu*, una aplicació mòbil per a organitzar viatges en grup.

Gràcies a *veelu*, els usuaris poden planejar les activitats que realitzaran durant un viatge, anotar en un checklist què portar a la maleta i tindre un control dels gastos que van tenint. A més d'aquestes ferramentes, *veelu* també permet als participants d'un viatge escriure els seus pensaments i experiències per mitjà d'entrades de cada dia, així com compartir fotografies, documents i altres arxius multimèdia.

Motivación, justificación y objetivo general

La idea de una herramienta para gestionar viajes en grupo surgió debido a una experiencia propia al hacer un viaje con amigos el pasado curso. Al ser estudiantes sin ingresos económicos, optamos por encargarnos nosotros mismos de planificar todos los aspectos del viaje para ahorrar el máximo dinero posible.

A la hora de organizar el mismo, hicimos uso de *Google Docs* para ir redactando en grupo toda la información vital sobre el viaje: el apartamento donde nos alojaríamos, los itinerarios de cada día, los vuelos, el dinero que costaba cada entrada o billete, etc.

Tras anotar definitivamente aquello que necesitábamos tener a mano, obtuvimos como resultado un documento desordenado y sin formato que dificultaba poder consultar cualquier dato de forma clara y rápida. Esto se tradujo en desorganización y en desconocimiento, pues la mayoría de los que participamos en el viaje no sabíamos, por ejemplo, qué actividades teníamos anotadas para cada día.

Por otro lado, durante el desarrollo del viaje, utilizamos otras aplicaciones como *WhatsApp* o las notas del móvil. La primera la usamos para comunicarnos y compartir fotografías, vídeos, documentos… entre nosotros. En cuanto a la aplicación de notas, la empleábamos para que no se nos olvidase información relevante o para apuntar el dinero que le debíamos a los demás.

Al final, fuimos capaces de planificar y llevar a cabo el viaje sin ningún problema. No obstante, tuvimos que invertir mucho tiempo y esfuerzo en redactar toda la información y clasificar nuestras fotografías, recuerdos… de una manera ordenada.

Esas son las principales consecuencias de emplear diferentes aplicaciones que no están diseñadas específicamente para la organización de un viaje: haber invertido más tiempo y esfuerzo del necesario.

Esto se podría haber evitado haciendo uso de algún sistema desarrollado con el fin de organizar viajes en grupo, una herramienta que centralizase todas las funcionalidades necesarias para facilitar y hacer mucho más cómodo todo este proceso, desde la organización hasta su finalización.

Desarrollar una herramienta con estas características es algo interesante y útil en una sociedad en la que, poco a poco, cada vez son más las personas que se animan a viajar. Tanto para estas personas que se están iniciando como para los que tienden a viajar con más frecuencia, sería de gran utilidad una aplicación móvil que les ayudara a organizarse de una manera muy intuitiva y fácil. Así también se evitaría tener que descargar numerosas *apps*[1] para conseguir el mismo objetivo.

Por otro lado, al ser una idea que ha surgido por un problema propio, me siento motivada y preparada para diseñar y desarrollar un proyecto que lo solucione. El haber vivido esa situación me permite poder cambiar la mentalidad de desarrolladora y ponerme en el lugar del usuario. Tengo claro qué es lo que se necesita y por eso quiero crear una herramienta que sea de gran utilidad para el mayor número de personas posible.

[1] Aplicación.

Agradecimientos

En primer lugar, quiero agradecer a mi tutor José Vicente por todo lo que he podido aprender de él en solo un año. Gracias por ser un profesor tan dedicado e innovador y por haberme guiado durante todo este curso.

En segundo lugar, me gustaría mencionar a todos mis amigos de Ingeniería Multimedia. He tenido la suerte estar rodeada de compañeros que me han hecho crecer tanto académicamente como en lo personal. Gracias por los momentos que hemos compartido estos cuatro años.

Por último, quiero dar las gracias a mi familia, a mi pareja y al resto de mis amigos por escucharme siempre, por animarme en mis malos momentos y por creer en mis posibilidades. Gracias por todo el apoyo que me habéis dado.

You need to be aware of what others are doing, applaud their efforts, acknowledge their successes and encourage them in their pursuits. When we all help one another, everybody wins.

– Jim Stovall

Who looks outside, dreams; who looks inside, awakes.

– Carl Jung

Índice de contenidos

Índice de figuras

Índice de tablas

1. Introducción

En la actualidad, son cada vez más las personas que se animan a viajar a menudo. Si se intenta analizar y definir un perfil de los individuos que realizan viajes, se llegará a la conclusión de que no hay uno concreto.

Las personas interesadas en viajar pueden ser desde familias que van a pasar sus vacaciones en el extranjero, empresarios que se ven obligados a viajar por su trabajo, jubilados que pasan su tiempo libre en nuevos lugares… hasta jóvenes que realizan viajes de fin de estudios.

No obstante, hay un problema que todos ellos comparten: planificar y realizar un viaje es un proceso tedioso al cual hay que dedicarle bastante tiempo si se quieren evitar complicaciones futuras. Si bien es cierto que también se puede delegar esta tarea a las agencias de viajes, son muchas las personas que prefieren organizar ellas mismas los viajes desde cero para ahorrar más dinero.

Si se trata del segundo caso, planificar un viaje es un proceso largo porque se han de tener en mente numerosos elementos. Es importante concretar la duración del viaje, el alojamiento, el transporte, qué hacer durante cada día, cuánto dinero va a costar cada actividad, qué cosas hay que llevarse en la maleta, cuál es el presupuesto aproximadamente… Y estos son solo unos pocos ejemplos de todos los factores a tener en cuenta.

Además, si se le añade la característica de que el viaje es en grupo y se han de tomar estas decisiones entre todos, esta tarea se vuelve más difícil y requiere más esfuerzo.

A la hora de anotar toda la información necesaria, se puede optar por el método tradicional de ir apuntándola en un papel o bien se puede emplear alguna tecnología que facilite esta tarea. Lo más común es hacer uso de alguna *app* móvil o página web.

Sin embargo, si se hace un análisis rápido de las aplicaciones ya existentes, se comprobará que pocas poseen las funcionalidades que se esperan. La mayoría se centran en una tarea específica, como planificar el itinerario o llevar un control de los gastos totales de un viaje. Y las que, en mayor o en menor medida, sirven para planificar varios aspectos de un viaje, suelen tener una interfaz anticuada, poco usable y no permiten la gestión colaborativa.

En definitiva, se puede concluir que la dificultad y el esfuerzo que supone planificar un viaje es un problema que afecta a la mayoría de las personas y que, a día de hoy, todavía no existe una solución óptima para solventarlo.

Puesto que es un problema muy extendido, desarrollar una herramienta que hiciese fácil, intuitivo y cómodo este proceso supondría una gran ayuda y una simplificación de la necesidad de organizar viajes en grupo.

Por este motivo, este Trabajo de Fin de Grado va a consistir en el diseño y desarrollo de esta herramienta para organizar viajes de forma colaborativa. En concreto, se tratará de una *app* móvil puesto que las personas pueden consultar de forma más rápida y sencilla este tipo de información mediante su *smartphone*[2].

La aplicación va a englobar todos los requisitos que se necesitan para tener ordenados los datos sobre un viaje. Entre estos requerimientos, se encuentra la planificación del itinerario por días, el control del presupuesto y de los gastos, la anotación de documentos y otro tipo de objetos que llevar al viaje y el intercambio de archivos multimedia, entre otros.

Como se ha mencionado anteriormente, si se trata de un viaje organizado por varias personas se complica enormemente esta tarea. Por esta razón, la *app* estará diseñada para favorecer también la organización colaborativa del viaje. Si bien habrá un administrador principal por cada viaje, el resto de los participantes tendrán la posibilidad de hacer modificaciones en la información de igual manera.

Durante el resto del documento, se profundizarán mucho más estos requisitos y se irá detallando todo el proceso de creación de esta *app* móvil, desde su diseño y especificación hasta su implementación y puesta en marcha.

[2] Teléfono inteligente con pantalla táctil y un robusto sistema operativo con el que los usuarios pueden conectarse a Internet, instalar aplicaciones, etc.

2. Estudio de viabilidad

Antes de pasar a otros apartados como la planificación o diseño, es vital hacer una investigación sobre el proyecto en sí mismo. Es el paso más importante antes de convertir la idea del producto en realidad. Además, este estudio puede servir en un futuro para tomar decisiones estratégicas.

En concreto, se ha optado por hacer un análisis DAFO y realizar un *Lean Canvas*. Por otro lado, también se hará un estudio exhaustivo de los posibles riesgos y los planes de contingencia.

2.1. Análisis DAFO

El análisis DAFO (de las iniciales de Debilidades, Amenazas, Fortalezas y Oportunidades), también conocido como FODA, es una metodología de estudio de la situación de un proyecto para poder tomar decisiones en un futuro [1]. Se basa en el análisis de características internas (Debilidades y Fortalezas) y de la situación externa (Amenazas y Oportunidades) en una matriz cuadrada como muestra la Figura 1.

Figura 1. Esquema de un análisis DAFO.
(Fuente: http://consultorioempresarial.blogsiga.net/analisis-dafo-una-herramienta-basica-para-emprendedores/)

Tras reflexionar sobre las características del proyecto, el análisis DAFO resultante es el siguiente:

- **Debilidades**:
    - Desconocimiento de las tecnologías a utilizar.
    - Inexperiencia desarrollando y manteniendo aplicaciones móviles.
    - Posibilidad de no saber resolver problemas relacionados con dichas tecnologías debido a la falta de experiencia.
    - Todo el diseño, creación y éxito de la aplicación recae en una sola persona.
    - Dificultad para llegar a un público amplio por la falta de recursos económicos.
    - Tiempo de desarrollo limitado a 300 horas.

- **Amenazas**:
    - Posibilidad de que el producto sea copiado.
    - Tener agujeros que afecten a la seguridad de la aplicación.
    - Existencia de otras *apps* similares.

- **Fortalezas**:
    - Aplicación completamente gratuita, a diferencia de las alternativas existentes.
    - Desarrollar el proyecto desde el principio de curso, teniendo margen de error por si surge algún inconveniente.
    - Conocer la mentalidad de los posibles usuarios ya que se ha experimentado su misma situación.
    - Saber organizar y planificar bien las tareas a realizar, así como repartir el tiempo disponible.
    - Experiencia con el autoaprendizaje de *frameworks*[3]/lenguajes de programación.
    - Revisión periódica de los avances del proyecto por parte de un tutor entendido en la materia.

- **Oportunidades**:
    - Es una aplicación que puede interesar a un gran número de personas, ya que viajar es una de las aficiones más extendidas.
    - Aprender conocimientos sobre el desarrollo de aplicaciones móviles.
    - Familiarizarse con las tecnologías a emplear.
    - Lograr un producto que contribuya a dar una buena imagen en un *portfolio*.

[3] Marco de trabajo. Esquema o estructura que se establece y que se aprovecha para desarrollar y organizar un *software* determinado.

- Dejar constancia de que estoy capacitada para desarrollar un proyecto de esta extensión desde cero.

2.2. Lean Canvas

El *Lean Canvas* es una herramienta para plasmar un modelo de negocio innovador y viable, propio de una *startup*[4], pues se centra más en el producto a desarrollar que en la propia empresa [2]. El *Lean Canvas* permite analizar el proyecto desde distintas perspectivas. En la parte derecha se refleja el mercado, mientras que en la izquierda se analiza el producto o servicio.

Al utiliza el *Lean Canvas* para la aplicación a desarrollar, se obtiene la Figura 2:

Figura 2. Cuadro para el análisis Lean Canvas.
(Fuente propia)

Con este cuadro se pueden visualizar rápidamente los puntos clave del producto de una forma simplificada, tanto para los involucrados en el proyecto como para las personas que no. No obstante, a continuación se detallan cada uno de estos bloques para una mejor explicación.

[4] Empresas que se encuentran en edad temprana o nueva creación y tienen posibilidades de crecimiento.

2.2.1. Segmento de clientes

- **Turistas**: personas que tienden a viajar con frecuencia por afición. Por lo general, suelen tener un nivel económico medio/alto. Les gusta indagar sobre los lugares que van a visitar antes de realizar el viaje en sí. La mayoría de las veces viajan en grupo y suelen tomar muchas fotografías.

- **Personas que viajan frecuentemente por negocios**: individuos cuyos empleos les exigen desplazarse a menudo. Necesitan tener anotadas todas las actividades y citas que han acordado de una forma concisa y clara.

- **Mochileros**: personas que deciden explorar múltiples lugares para tomar un respiro de su vida cotidiana. Aunque toman decisiones de forma espontánea, les gusta inmortalizar momentos con fotografías o anotando sus experiencias diarias.

- ***Influencers***: jóvenes que tratan de darse a conocer a través de las redes sociales y obtener beneficios económicos por ello. Su característica principal es que comparten casi cualquier aspecto de su vida privada por Internet. Esto incluye numerosas fotografías de sus viajes y opiniones sobre el mismo.

- **Estudiantes de más de 18 años (*early adopters*)**: gente que ya tiene la edad suficiente para viajar, pero no tiene ingresos económicos. Prefieren planificar todo ellos mismos desde cero, ya que tienen la libertad de establecer las actividades que prefieren y, sobre todo, porque ahorran más dinero de esta forma. Están acostumbrados al uso de las nuevas tecnologías, por lo que son más propensos a buscar nuevas aplicaciones que les solucionen o faciliten sus tareas.

Arquetipos

- Raquel, joven de 20 años que estudia en la Universidad de Alicante. Sus amigos y ella quieren planear un viaje barato a Bélgica porque no tienen mucho dinero. Raquel es una persona detallista y ordenada, y quiere anotar las actividades de cada día para tener claro cuáles son los planes establecidos.

- Trini, madre de 55 años que suele viajar cada verano con su pareja y sus amigos. Debido a su trabajo, tiene un nivel económico medio, pero no dispone de mucho tiempo libre. Por ello, contrata agencias para que se encarguen de toda la planificación de los viajes. Sin embargo, es algo olvidadiza, por lo que le gusta tener anotado qué tiene llevarse en la maleta y tener a mano todos los billetes y documentos que necesita. Además, le gusta tomar muchas fotos y tener guardadas las que han hecho sus amigos.

2.2.2. Problemas

- Falta de herramientas concretas para organizar viajes de manera colaborativa. Del análisis anterior se puede deducir que la mayoría de las personas viajan en grupo. No obstante, en el mercado actual no hay muchas herramientas que faciliten la organización colaborativa.

- Las aplicaciones existentes son muy específicas y no poseen todos los requisitos para planificar varios aspectos de los viajes. Por ejemplo, en la *Play Store*[5] hay publicadas *apps* para controlar los gastos, conocer lugares que visitar, obtener información de los vuelos… pero casi ninguna de ellas permite planificar múltiples aspectos.

- Las alternativas actuales poseen interfaces anticuadas y/o muy complejas. No son usables. Esto provoca que el usuario no llegue a utilizar la aplicación o se frustre durante el proceso de organizar la información. También dificulta visualizar luego estos datos.

2.2.3. Soluciones

- Centralizar todos los requisitos necesarios para organizar un viaje en una sola *app*. Así se evita el uso de varias aplicaciones que cubran las distintas necesidades. Además, esto facilita y agiliza la planificación.

- Ofrecer una interfaz usable e intuitiva. Dado que la aplicación también será interesante para gente más mayor, es imprescindible que la interfaz sea amigable y fácil de usar.

[5] Plataforma de distribución digital de *apps* móviles para dispositivos con sistema operativo *Android*.

Asimismo, dada la cantidad de información que la *app* pretende mostrar, la interfaz debe estar bien seccionada y mostrar todos los datos de forma clara.

- Permitir la planificación colaborativa. La tarea de organizar un viaje no suele recaer en solo una persona, por lo que el resto de participantes deben poder añadir, editar y eliminar cualquier tipo de información sobre el viaje.

2.2.4. Proposición de valor única

La propuesta para solucionar los problemas comentados anteriormente consiste en una aplicación móvil, la cual permitirá planificar los aspectos más importantes de un viaje, admitiendo también hacerlo de forma colaborativa.

2.2.5. Canales

- **Redes sociales**: mediante el uso de cuentas públicas en *Twitter*, *Instagram*, *Facebook* y otros sitios web, se intentará dar a conocer la *app* al mayor número de personas posible.

- **Boca a boca**: también llamado *marketing*[6] de recomendación [3]. La *app* podrá conseguir nuevos usuarios mediante la sugerencia de personas que estén satisfechas con su uso. Si estos usuarios consideran que la aplicación solventa o reduce alguno de sus problemas, es probable que lo recomienden a las personas de su entorno.

2.2.6. Flujo de ingresos

En principio, la aplicación no va a buscar ningún beneficio económico al tratarse de un proyecto educativo que no contempla la necesidad de asegurar un flujo de ingresos. No obstante, no se descarta incluir el modelo de negocio *freemium*[7] si la aplicación consigue tener éxito y se decide añadir más funcionalidades.

[6] Conjunto de estrategias empleadas para comercializar un producto y estimular su demanda.
[7] Modelo de negocio que funciona ofreciendo servicios básicos gratuitos, mientras se cobra dinero por otros servicios más avanzados o especiales.

2.2.7. Estructura de costes

- **Recursos humanos**: este punto hace referencia al sueldo de las personas involucradas en el proyecto. Como se trata del Trabajo de Fin de Grado, que es individual, solo supondría coste el sueldo de una persona.

- **Publicación en *Google Play Store***: para subir una *app* a esta plataforma y así permitir que otros usuarios la descarguen, es necesario adquirir una licencia como desarrollador. Esta licencia tiene un coste de 25$. También se ha planteado la posibilidad de subir la aplicación a la *App Store*, de *Apple*, pero debido al coste anual de 99$ y a no tener fuentes de ingresos, se ha descartado esta opción.

2.2.8. Métricas clave

- **Puntuación en la *Play Store***: la satisfacción media de los usuarios, puntuada del 1 al 5. Permite un *feedback*[8] directo de los usuarios, así como comentarios críticos que pueden ayudar a mejorar la aplicación.

- **Retención**: lo propenso que es el usuario a volver a usar la *app*. Es algo a tener en cuenta, ya que cuesta más trabajo atraer a nuevos clientes que mantener a los actuales.

- **Número de usuarios activos**: conocer cuántos usuarios, de la totalidad de aquellos que tienen descargada la *app*, hacen un uso frecuente de la misma.

- ***Churn rate***: es un indicador de la tasa de pérdida de clientes. Son los usuarios que dejan de utilizar la aplicación.

2.2.9. Ventaja injusta

No hay ningún factor por el cual esta aplicación no pueda ser copiada. Puesto que no hace uso de datos de terceros, no se pueden obtener licencias o información exclusiva. Quizás se podría

[8] En este caso, la opinión que tienen los usuarios de la aplicación.

considerar como ventaja injusta el hecho de que la aplicación será totalmente gratuita y la competencia, si desea obtener beneficio, tendrá que añadir anuncios o limitar algunas funcionalidades a cambio de una suscripción.

2.3. Análisis de riesgos

El objetivo de este apartado es enumerar los distintos contratiempos e inconvenientes que pudiesen surgir durante la realización del TFG. Al ser un proyecto equivalente a 300 horas, es recomendable tener en cuenta cualquier riesgo que pueda afectar de forma negativa al desarrollo de este.

En la Tabla 1, se clasifican y enumeran cada uno de estos riesgos según su tipo, se indica la probabilidad de que estos ocurran y cómo de grave sería su impacto (ordenados de mayor a menor impacto).

En la columna de la probabilidad, cada expresión hace referencia a un porcentaje: muy baja (menor del 10%), baja (entre el 10% y el 25%), moderada (entre el 25% y el 50%), alta (entre el 50% y el 75%) y muy alta (mayor del 75%).

Tabla 1. Clasificación de riesgos.

Riesgo	Probabilidad	Efecto
Tecnología		
Problemas técnicos con las infraestructuras seleccionadas para el desarrollo.	Baja	Catastrófico
Avería grave del portátil u otra causa que se traduzca en la necesidad de comprar otro.	Baja	Serio
Que los servicios de APIs[9] de terceros dejen de ser accesibles.	Baja	Serio
Que el rendimiento de la aplicación no sea el esperado.	Moderada	Tolerable
Avería leve del portátil.	Alta	Tolerable

[9] Interfaz de programación de aplicaciones. Conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el *software* de las aplicaciones.

Personal		
Sufrir un accidente grave.	Muy baja	Catastrófico
Sufrir estrés causado por la falta de tiempo o por acumulación de problemas.	Alta	Serio
Problemas o dificultad a la hora de planificar las subtareas a realizar.	Moderada	Tolerable
Contraer una enfermedad.	Muy alta	Tolerable
Herramientas		
Pérdida completa o parcial de fuentes y documentos.	Muy baja	Catastrófico
Fallos en los servicios web usados para la monitorización de tareas y el tiempo de trabajo.	Muy baja	Insignificante
Que el IDE[10] a utilizar de fallos o deje de ser gratuito.	Muy baja	Insignificante
Requerimientos		
Cambios drásticos en los requerimientos que supongan modificaciones significativas en la base de datos y/o la API.	Baja	Serio
Estimación		
Infraestimar o sobreestimar el tiempo necesario para implementar la *app*.	Alta	Serio
Infraestimar o sobreestimar el tiempo necesario para redactar los apartados de la memoria del TFG.	Alta	Tolerable
Otros		
Que otra persona o equipo implemente, para el problema de planificar viajes en grupo, una solución similar o mejor que la diseñada en este TFG.	Moderada	Serio

A continuación, se muestra una tabla en la que, en relación con los riesgos enumerados anteriormente, se describe la estrategia a seguir para prevenir, minimizar y/o ponerse en el peor de los casos (plan de contingencia) si los riesgos se cumplen.

[10] Aplicación informática que proporciona servicios integrales para facilitar el desarrollo de *software*.

Tabla 2. Estrategias para solucionar los riesgos.

Riesgo	Estrategia
Tecnología	
Problemas técnicos con las infraestructuras de desarrollo.	Prevención: ir haciendo pequeñas pruebas para confirmar que no surgen dificultades cuando se vaya a desplegar la *app*. Minimización: apoyarse en la documentación, en los foros o contactar con gente especializada para resolver problemas de gravedad.
Avería grave del portátil u otra causa que se traduzca en la necesidad de comprar otro.	Minimización: hacer uso de un portátil antiguo o de los ordenadores de la facultad hasta comprar uno nuevo.
Que los servicios de APIs de terceros dejen de ser accesibles.	Minimización: optar por otra de las alternativas del mercado.
Que el rendimiento de la aplicación no sea el esperado.	Prevención: informarse acerca de las mejores prácticas para mantener optimizada la aplicación. Minimización: hacer uso de herramientas para analizar el rendimiento de la *app* que permitan encontrar el foco del problema.
Avería leve del portátil.	Prevención: utilizar antivirus y llevar un mantenimiento adecuado del ordenador personal. Minimización: buscar solución al problema concreto a través de Internet. En caso de no poder arreglarlo, acudir a algún especialista informático.
Personal	
Sufrir un accidente grave.	Prevención: establecer desde el principio una planificación adecuada, en la cual se incluyan un par de semanas de margen para que cualquier tipo de inconveniente no afecte a los tiempos. Plan de contingencia: presentar lo desarrollado hasta la fecha o aplazar la entrega para el curso siguiente.
Sufrir estrés causado por la falta de tiempo o por acumulación de problemas.	Prevención: planificar con antelación las tareas a realizar, establecer fechas concretas y repartir el trabajo de forma en la

	que se pueda trabajar poco a poco en el TFG sin descuidos ni agobios. Minimización: priorizar tareas y replantear la planificación. Decidir si es necesario quitar algún requisito especificado.
Problemas o dificultad a la hora de planificar las subtareas a realizar.	Prevención: a partir de cada uno de los apartados, hacer una lista de subtareas e intentar fijar un periodo aproximado para su realización.
Contraer una enfermedad.	Minimización: continuar con la realización del TFG a un ritmo más lento o completar tareas sin mucha dificultad.
Herramientas	
Pérdida completa o parcial de fuentes y documentos.	Prevención: tener más de un sistema de copias de seguridad y hacer copias con frecuencia. Minimización: hacer uso de la última versión que haya en la nube (si se puede acceder) o en local.
Fallos en los servicios web usados para la monitorización de tareas y el tiempo de trabajo.	Prevención: hacer capturas y descargar informes que reflejen el estado de las tareas y las horas dedicadas a las mismas. Minimización: seleccionar otras alternativas y hacer una estimación del tiempo empleado hasta la fecha.
Que el IDE a utilizar de fallos o deje de ser gratuito.	Minimización: investigar y comparar otros IDE gratuitos y fiables disponibles en el mercado y decantarse por uno de ellos.
Requerimientos	
Cambios drásticos en los requerimientos que supongan modificaciones significativas en la base de datos y/o la API.	Prevención: tener una base de datos y una API bien diseñadas y estructuradas para que modificar, añadir o eliminar requisitos no suponga rediseñar todo.
Estimación	
Infraestimar o sobreestimar el tiempo necesario para implementar la *app*.	Prevención: informarse, ver tutoriales y hacer pruebas para comprender bien el funcionamiento de las tecnologías para así poder hacer una estimación correcta. Minimización: replantear y subdividir las tareas a realizar.
Infraestimar o sobreestimar el tiempo necesario para redactar los apartados de la memoria del TFG.	Prevención: mirar ejemplos de TFG de cursos anteriores y hacer los cursos relacionados con información sobre el mismo, para hacerse una idea de los contenidos y de la extensión de los apartados.

	Minimización: redistribuir el tiempo que se había estimado para la redacción de los distintos capítulos.
Otros	
Que otra persona o equipo implemente, para el problema de planificar viajes en grupo, una solución similar o mejor que la diseñada en este TFG.	Plan de contingencia: seguir trabajando en el proyecto como se tenía previsto. Estudiar la *app* para conocer sus debilidades y fortalezas, y sacar alguna conclusión que se pueda aplicar al proyecto.

3. Planificación

Organizar todos los apartados a realizar en un proyecto es una tarea imprescindible para obtener mejores resultados. Una buena planificación permite gestionar el tiempo disponible, priorizar acciones y reaccionar ante imprevistos, entre otros.

Además, tras finalizar todo el trabajo, también es de gran utilidad para comparar las estimaciones previas al proyecto con los resultados reales.

La planificación del TFG estará dividida según los apartados del mismo, tal y como se muestra en la Tabla 3.

Tabla 3. Planificación temporal del TFG.

Contenidos	Tiempo total	Fecha límite fin
Motivación, justificación, objetivo general Introducción Estudio de viabilidad Planificación Estado del arte	2 meses	15 noviembre
Objetivos Metodología Análisis y especificación	1 mes	20 diciembre
Diseño	2 meses	1 marzo
Implementación	3 meses	1 junio
Pruebas y validación Resultados Conclusiones y trabajo futuro Referencias, bibliografía y apéndices Agradecimientos Citas Índices Resumen	1 mes	18 julio

En resumen, se pretende compaginar la realización del Trabajo de Fin de Grado con el resto de las asignaturas de cuarto curso. Por ello, aunque a veces no sea posible dedicarle unas horas diarias al trabajo individual, se procurará hacer avances semanales de forma constante. El propósito es desarrollar el TFG poco a poco sin descuidarlo en ningún momento.

Si se cumple la planificación estimada, se tendría todo el proyecto terminado para la convocatoria de julio. Aunque también está la posibilidad de entregar el TFG en la convocatoria de mayo, lo más probable es que no esté finalizado para esa fecha debido a factores externos que pueden retrasar su realización.

El principal de ellos es la entrega final y presentación del proyecto en grupo desarrollado durante todo el curso. Esta entrega se realiza a finales de mayo, así que durante los meses previos todo el tiempo disponible se invertirá en acabar y pulir dicho producto.

Por lo tanto, se pretende dedicar el primer cuatrimestre a los apartados iniciales, donde se define la idea y los objetivos del proyecto. Durante las vacaciones de Navidad y los meses posteriores, se pasará a la realización del diseño. Luego, durante los siguientes meses hasta el principio del verano, se implementará la aplicación. Por último, se le dedicará un mes al análisis del resultado final y a completar el resto de la memoria.

4. Estado del arte

Este apartado tiene el objetivo de hacer una investigación profunda de la problemática del proyecto a desarrollar para convertirse en un experto sobre el tema, y así poder ofrecer soluciones adecuadas.

Por ello, en el estado del arte se indaga en las herramientas existentes que tratan de aportar solución a esta problemática y se hace un estudio de las posibles tecnologías a utilizar, para elegir las más adecuadas al proyecto.

4.1. Soluciones existentes

En el mercado actual, ya existen aplicaciones que intentan dar solución al problema de organizar de viajes en grupo o a alguno de sus subproblemas. Por lo tanto, se ha hecho una pequeña selección y análisis de aquellas cuyo estudio sería de gran utilidad antes de comenzar con el desarrollo del proyecto.

Para filtrar las *apps* a estudiar, se han tenido en cuenta varios factores. En primer lugar, se ha optado por analizar únicamente aplicaciones móviles, puesto que, si se viaja y se desea tener toda la información a mano, utilizar una *app* siempre será más cómodo que usar una web. Por lo tanto, se ha hecho uso de la *Play Store* para indagar en este tipo de herramientas.

A la hora de hacer la búsqueda, se ha utilizado un navegador web que permitiese la navegación privada para que, al buscar en la *Play Store*, no tuviese en cuenta factores como la ubicación o las preferencias asociadas a una cuenta de *Google*.

Por último, los términos a emplear para la búsqueda han sido: "*trip planner*", "planear viaje" … y luego se ha ido especificando más, utilizando frases como "organizar viajes en grupo" o "*trip planner group*". Para las *apps* especializadas, se ha buscado "control gastos viaje" y "maleta viaje", entre otros. Además, solo se han tenido en cuenta *apps* que tuviesen, como mínimo, una valoración de 4 estrellas sobre 5 por parte de los usuarios, ya que su satisfacción con la aplicación es un factor clave.

En definitiva, tras aplicar todas estas características y filtros de búsqueda, se han seleccionado y analizado las aplicaciones móviles que se muestran a continuación:

4.1.1. SaveTrip

Puntuación de 4,2/5 en la *Play Store* [4]. Tras crear un viaje (indicando las fechas y el lugar), esta *app* permite establecer itinerarios personalizados. Además, tiene otros apartados como el control del presupuesto, un *checklist*[11], notas y también permite fotos (solo este tipo de archivo), entre otras funcionalidades. Si bien es cierto que la *app* recoge la mayoría de aspectos relevantes de un viaje, no ofrece la posibilidad de planear de forma colaborativa.

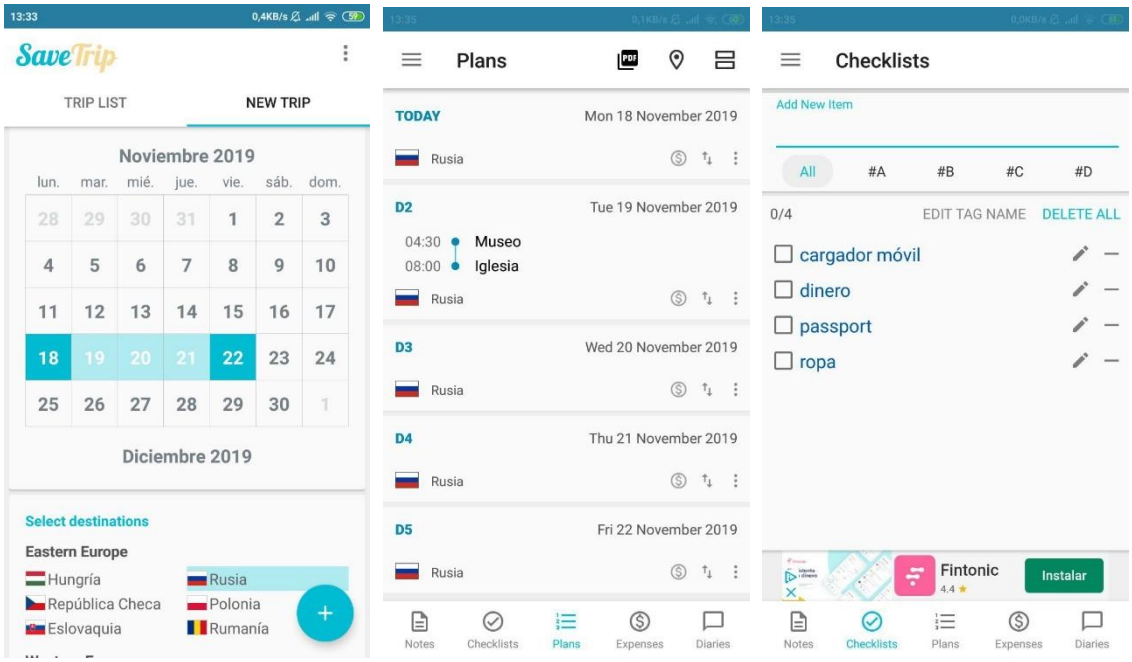


Figura 3. Interfaces de SaveTrip.
(Fuente propia)

4.1.2. TripIt

Puntuación de 4,5/5 en la *Play Store* [5]. Al igual que sucede con *SaveTrip*, esta aplicación permite crear itinerarios de viaje personalizados. La principal diferencia respecto a la anterior, es que *TripIt* sí permite la planificación del viaje en grupo, enviando un e-mail con una invitación a los

[11] Listado general de aquellas cosas más necesarias e imprescindibles que hay que llevar en la maleta.

participantes del viaje. También se centra en mostrar información de los vuelos escogidos, como la hora a la que sale el avión, la duración, el asiento…

Sin embargo, no tiene ninguna funcionalidad relacionada con el control de los gastos ni permite escribir notas o utilizar *checklists*, así como la subida de archivos que no sean fotografías (como PDFs, vídeos…). Además, su diseño es algo confuso, lo cual perjudica notablemente en la experiencia de usuario.

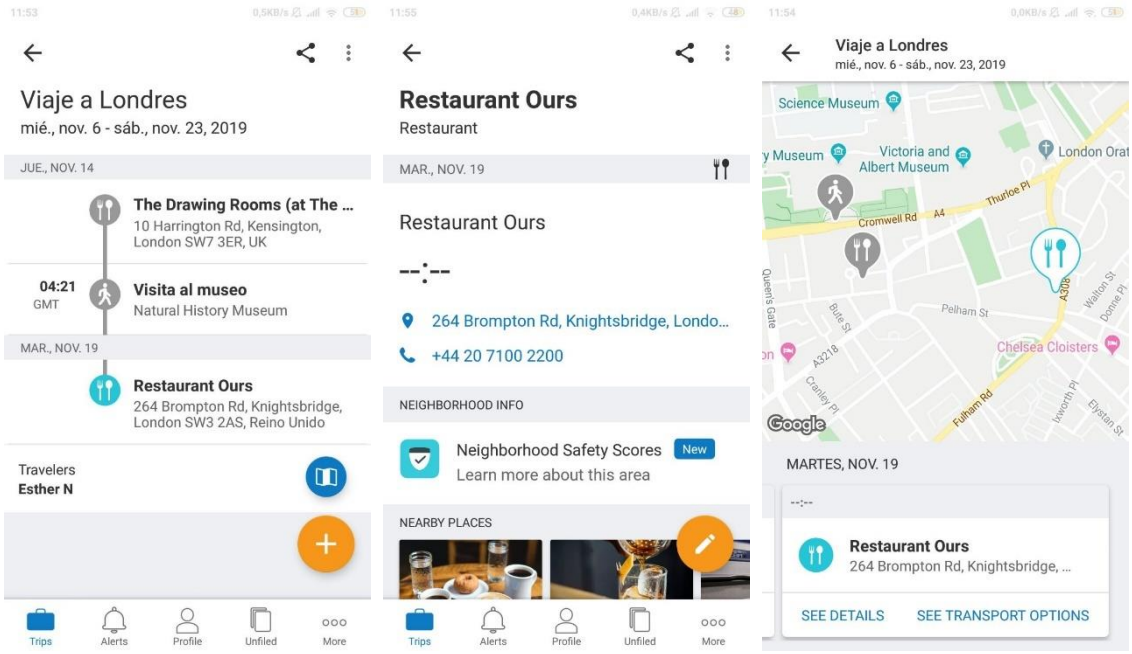


Figura 4. Interfaces de TripIt.
(Fuente propia)

4.1.3. HOTSGO PLAN

Puntuación de 4,4/5 en la *Play Store* [6]. Comparte muchas de las características de las dos *apps* ya analizadas. Esta aplicación también maneja y permite personalizar los itinerarios, se puede buscar a través de un mapa, controla el dinero gastado, incluye una sección para la maleta y permite la organización de forma colaborativa, entre otros.

Como puntos en contra, aunque tiene un mejor diseño de interfaz en comparación a las anteriores *apps*, todavía sigue siendo complicado entender de forma rápida su funcionamiento, pues incorpora muchos botones cuya funcionalidad no está muy clara hasta que los pruebas. Además, esta aplicación, originalmente coreana, traduce todo su contenido al español de una forma incorrecta o muy poco natural, lo cual dificulta a veces entender su funcionamiento.

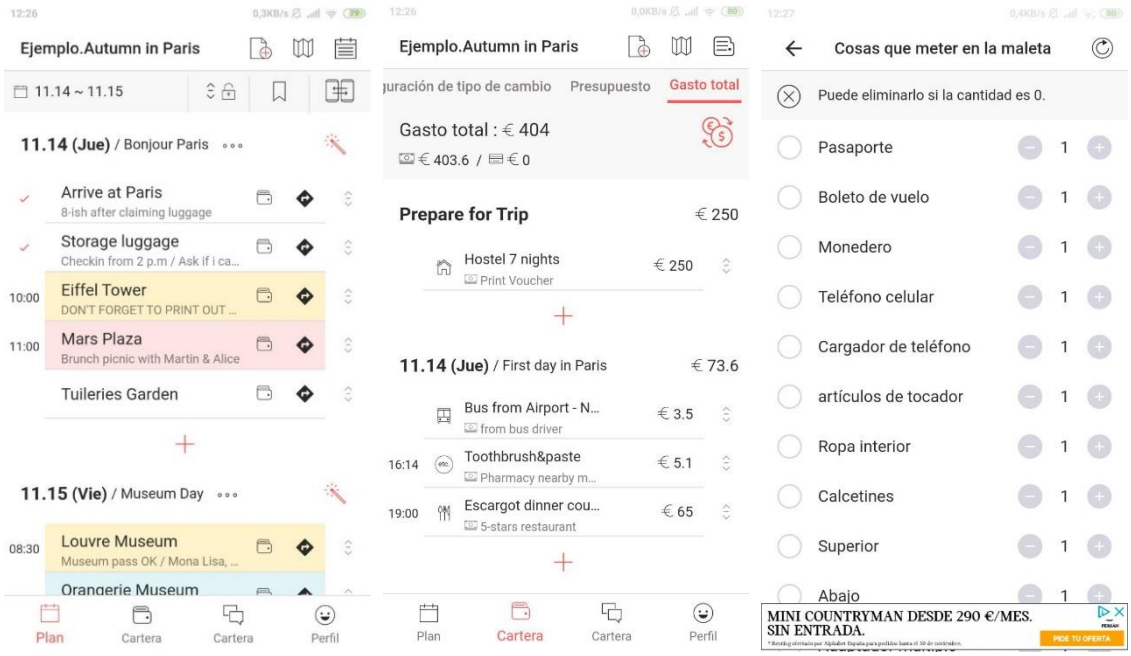


Figura 5. Interfaces de HOTSGO PLAN.
(Fuente propia)

4.1.4. Lambus

Puntuación de 4,5/5 en la *Play Store* [7]. *Lambus* es la aplicación que mejor resuelve la problemática de organizar viajes en grupo. Posee numerosas funcionalidades como las mencionadas en el resto de *apps*, como definir itinerarios, uso de un mapa, controlar el presupuesto, subir fotografías... y permite que se haga en grupo.

Sin embargo, la manera en la que esta aplicación ha estructurado todas estas funcionalidades es completamente diferente. Con una interfaz muy minimalista y cuidada, el uso de esta herramienta es prácticamente intuitivo.

Como puntos en contra, se echa en falta alguna funcionalidad más especializada, como el uso del *checklist* (para crear una lista de cosas que llevar) o también la poca personalización de las actividades dentro del itinerario.

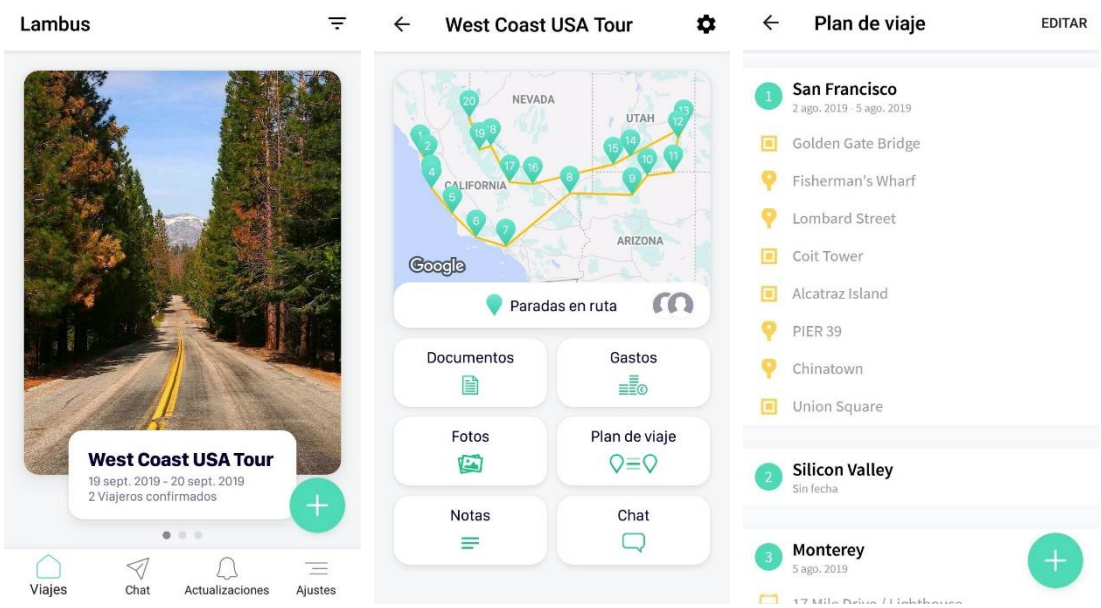


Figura 6. Interfaces de Lambus.
(Fuente propia)

4.1.5. TravelSpend

Puntuación de 4,6/5 en la *Play Store* [8]. Esta aplicación trata de aportar solución al subproblema de la gestión del presupuesto destinado a un viaje. *TravelSpend* lleva un recuento del gasto total y calcula el gasto promedio por día. También se puede ver cada uno de los gastos y filtrarlos según categorías para ver estadísticas.

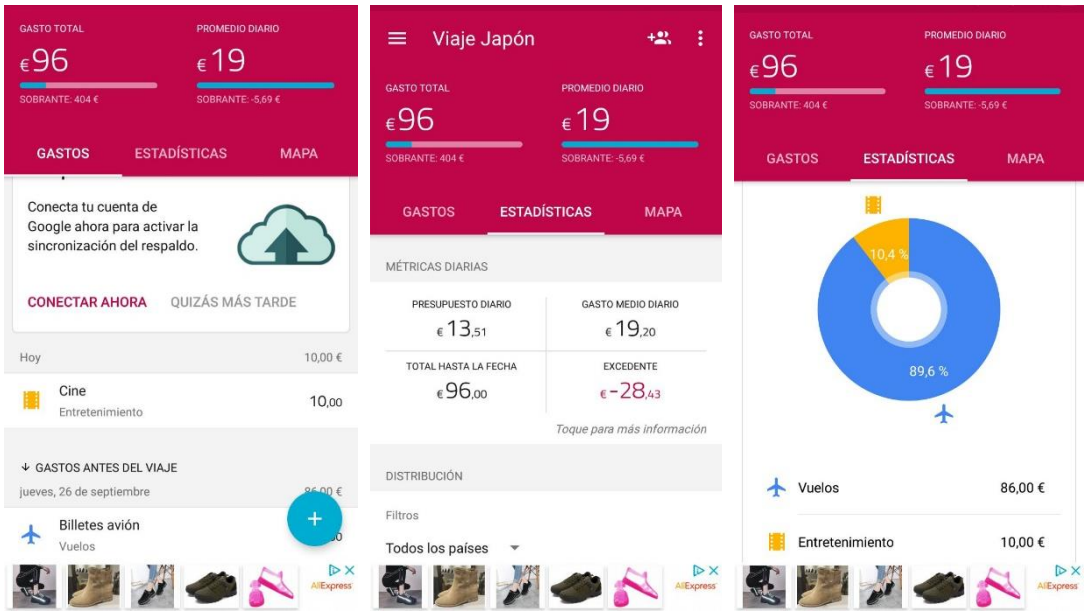


Figura 7. Interfaces de TravelSpend.
(Fuente propia)

4.1.6. PackPoint

Puntuación de 4,7/5 en la *Play Store* [9]. Esta *app* móvil tiene el objetivo de ayudar a los viajeros a la hora de crear una lista de equipaje. Para ello, hay que introducir unos parámetros básicos (lugar, duración del viaje, fecha…) y alguna de las actividades o características del mismo (se va a ir a la playa, se va a practicar deporte…). A partir de ahí, *PackPoint* tendrá en cuenta dicha información y la previsión del tiempo, y generará una lista personalizada de cosas que son necesarias para el viaje.

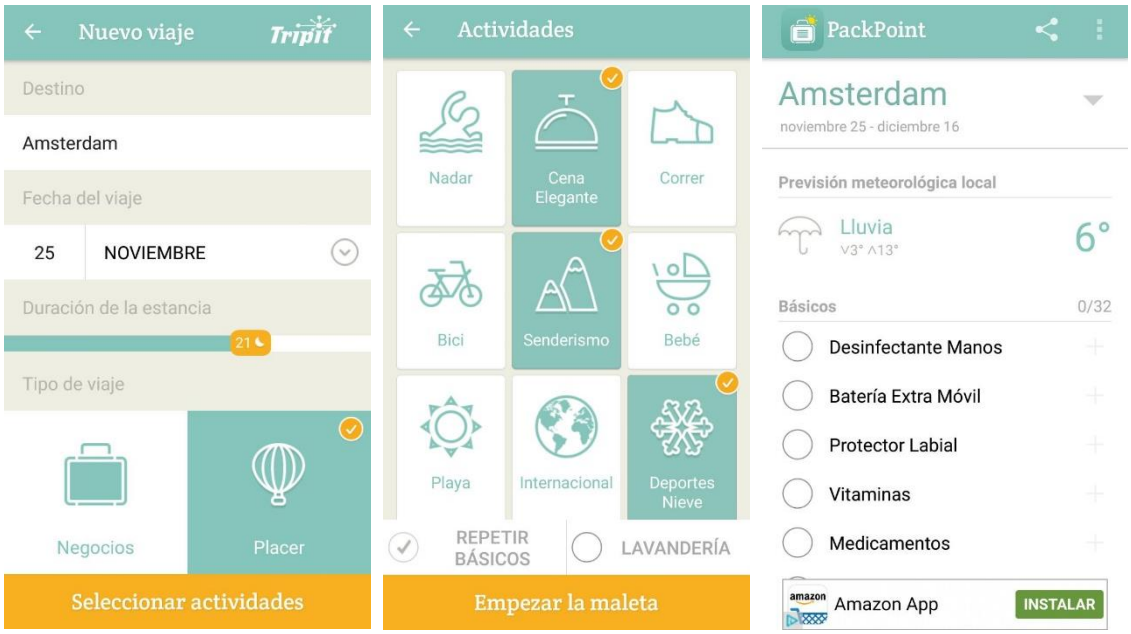


Figura 8. Interfaces de PackPoint.
(Fuente propia)

4.1.7. Conclusiones

Tras hacer un análisis de todas estas *apps*, se han deducido varios factores a tener en cuenta. El primero es la importancia de un buen diseño de interfaz. Al experimentar con las diferentes aplicaciones, a veces ha sido complicado averiguar cómo realizar ciertas funcionalidades o para qué servían ciertos botones. Además, mirando las valoraciones medias de las *apps* se observa que, a mejor interfaz, mayor satisfacción por parte de los usuarios.

Por otro lado, tras estudiar estas soluciones se han obtenido requisitos que deberían ser básicos, pues aparecen en todas o la mayoría de aplicaciones para organizar viajes. Estos son la creación

de itinerario, búsqueda a través de un mapa, control del presupuesto, *checklist* para preparar la maleta y poder subir archivos multimedia (fotos, PDFs…).

4.2. Tecnologías para el desarrollo

Antes de comparar tecnologías, es importante tener claras las diferencias entre las aplicaciones nativas y las híbridas [10]. Las *apps* nativas son aquellas que han sido desarrolladas específicamente para un sistema operativo móvil, como pueden ser *Android*, *iOS* o *Windows*. Por otro lado, las híbridas son un producto que funciona para distintos sistemas operativos.

En cuanto a las aplicaciones nativas, al desarrollarse para sistemas específicos, tienen un mejor rendimiento, mayor velocidad, un acabado más natural y mejor experiencia de usuario, entre otras ventajas.

Antes de desarrollar una *app* en nativo, primero se ha de tener claro a qué sistema va dirigido, pues cada uno de ellos tiene su lenguaje de programación y su entorno de desarrollo. Por ejemplo, si se quiere desarrollar una *app* para *Android*, es necesario programar en *Java* o *Kotlin* en *Android Studio*. Si se desea desarrollar para *iOS*, se tendría que trabajar con *Objective-C* o *Swift* en su IDE *AppleXCode*.

El principal problema de desarrollar en nativo es que, para cada sistema operativo, hace falta desarrollar un código completamente distinto. Esto provoca que se invierta mucho tiempo en la implementación, en el aprendizaje de los diferentes lenguajes de programación y también dificulta el mantenimiento del código.

La otra opción consiste en desarrollar una *app* híbrida. La principal ventaja que aporta es que se puede escribir el código de la aplicación una sola vez y utilizarlo para todas las plataformas que se prefieran. Es mucho más fácil de crear, controlar y mantener, y se ahorra tiempo y esfuerzo. En definitiva, el desarrollo híbrido soluciona todos los problemas que se producen al trabajar en una *app* nativa.

No obstante, esta opción también tiene sus desventajas. Algunas de ellas son las dependencias con diferentes librerías y *frameworks*, el rendimiento (suele ser más bajo) y que no siempre

ofrecen la mejor experiencia de usuario. Aun así, tras comparar ventajas y desventajas de ambas alternativas, para este TFG se ha optado por desarrollar una *app* híbrida.

A continuación, se enumera y se profundiza sobre los *frameworks* más relevantes de la actualidad para desarrollar este tipo de aplicaciones:

4.2.1. Ionic

Uno de los *frameworks* más populares y utilizados hasta la fecha [11]. Es gratuito y *open source*[12]. Permite desarrollar aplicaciones móviles y de escritorio utilizando HTML, CSS y *JavaScript*, es decir, tecnologías web. Está construido con *Apache Cordova* y librerías o *frameworks* como *Angular*, *React* o *Vue*, aunque también permite usar *JavaScript* puro.

Ionic se centra en la interacción con la interfaz de usuario, el estilo y el acabado. Para ello, pone a disposición numerosos componentes y temas que permiten a los desarrolladores implementar interfaces modernas. Además, estos se adaptan a los estilos propios de cada plataforma.

La principal ventaja es que, si uno está familiarizado programando con las tecnologías web, es muy fácil aprender a trabajar con *Ionic*. Esto ahorra mucho tiempo y esfuerzo. Por otro lado, tiene una extensa documentación y una gran comunidad de desarrolladores que participan activamente en sus foros.

Por otro lado, se tiene la desventaja de que una aplicación desarrollada con *Ionic* va a ser menos eficiente que una *app* nativa. También hay que recalcar que *Ionic* no es la mejor opción a la hora de implementar aplicaciones que empleen gráficos muy potentes.

4.2.2. React Native

Este *framework* desarrollado por *Facebook* está basado en *React* y *Javascript* [12]. A diferencia de la mayoría de *frameworks*, como el anteriormente mencionado *Ionic*, este no hace uso de

[12] *Software* cuyo código fuente y otros derechos son publicados bajo una licencia de código abierto o forman parte del dominio público.

WebViews[13], sino que utiliza componentes nativos internamente. Para construir las interfaces de usuario, utiliza JSX, una fusión de *JavaScript* con XML. Lo que hace *React Native* es renderizar, mediante componentes nativos, la aplicación en el lenguaje específico de la plataforma: *Swift* (*iOS*) o *Java* (*Android*).

Entre las ventajas de utilizar *React Native* está la característica llamada "*hot reloading*", que permite una visualización rápida de cómo quedaría la aplicación en *iOS* y *Android*. Además, al trabajar sobre controles y módulos nativos, este *framework* mejora el rendimiento.

La principal desventaja a tener en cuenta es que es un *framework* que todavía necesita madurez, puesto que es relativamente nuevo. Por otro lado, se necesitan mantener tres entornos diferentes (*React*, *iOS* y *Android*), lo que se traduce en la necesidad de contratar a programadores específicos de estas plataformas. Ambos problemas han provocado que compañías como *Airbnb* decidiesen dejar de utilizar *React Native* para el desarrollo de sus aplicaciones [13].

4.2.3. Flutter

Flutter es un *framework* desarrollado por *Google* con el objetivo de crear aplicaciones móviles rápidas y atractivas visualmente [14]. Al igual que los anteriores, es gratuito y *open source*. No obstante, la principal característica que le diferencia del resto de *frameworks* es que *Flutter* compila completamente las *apps* en código nativo.

Al igual que *React Native*, *Flutter* también permite ver cómo sería el resultado final de la *app* en las distintas plataformas. También hay que destacar la velocidad de sus aplicaciones resultantes, alcanzando 60 FPS[14] o la variedad de componentes que hay en el catálogo de *Flutter*.

Sin embargo, este *framework* requiere el aprendizaje de un nuevo lenguaje de programación, llamado *Dart*. Aunque este lenguaje orientado a objetos sea relativamente fácil de entender, sigue siendo una desventaja tener que invertir tiempo en aprender cómo utilizarlo. Al mismo tiempo, al ser un *framework* nuevo en el mercado, no posee una comunidad amplia, lo cual es un problema si se necesita resolver alguna duda en Internet.

[13] Componente del sistema con la tecnología de *Chrome* que permite a las aplicaciones de *Android* mostrar contenido web.
[14] *Frames* por Segundo.

4.2.4. Xamarin

Tras la compra por parte de *Microsoft* en 2016, *Xamarin* se convirtió en *open source* y fue integrado en la plataforma .NET [15]. Este *framework* está basado en C# y compila a código nativo para las distintas plataformas.

Lo mejor de *Xamarin* es que iguala el rendimiento de las aplicaciones nativas con poca o ninguna diferencia. También hace uso de elementos de interfaz nativos. Es importante destacar el fácil mantenimiento, pues si se desea modificar una *app* con *Xamarin* solo es necesario cambiar el archivo fuente, eliminando la necesidad de modificar individualmente los códigos de las distintas plataformas.

Por otro lado, puede ser complicado el uso de alguna librería *open source*, las *apps* suelen tener un gran tamaño y supone un enorme coste económico para grandes empresas, ya que que se deben comprar licencias de su IDE (*Microsoft Visual Studio*).

4.2.5. Conclusión

Tras comparar el desarrollo nativo con el híbrido y teniendo en cuenta la aplicación a implementar, se ha decantado por el desarrollo híbrido. El principal motivo es la posibilidad de desplegar la aplicación en otras plataformas en un futuro, por lo que poder escribir el código solo una vez y tenerlo disponible para varios entornos es algo primordial.

En concreto, se ha escogido el *framework Ionic* (basado en *Angular*). Esto se debe a que utiliza tecnologías web con las que se tiene bastante experiencia, y también por su extensa documentación y gran comunidad, que aportan la seguridad de poder resolver cualquiera de los problemas que pudiesen surgir durante el desarrollo.

4.3. Backend

La decisión de cómo implementar el almacenamiento, seguridad y la lógica de una aplicación no debe tomarse a la ligera. Todo esto es lo que se conoce como *backend*. Debido a su importancia,

es necesario analizar adecuadamente las características de la aplicación y las distintas opciones que existen para desarrollar el *backend*.

La aplicación a implementar servirá para planificar viajes en grupo. Hay que hacer hincapié en esta última característica: se trata de una *app* colaborativa. Por lo tanto, es fundamental que los datos entre los distintos usuarios estén sincronizados en tiempo real. También se deberá poder almacenar archivos multimedia, como fotos o vídeos, que suelen ocupar un espacio considerable.

Cuando se habla del *backend* de una *app*, se hace referencia a un servidor para aplicaciones móviles donde se almacena y se ordena la información. En la actualidad, existen dos formas comúnmente empleadas al desarrollar un servidor para *apps*: crear uno propio o utilizar un servicio existente.

La primera posibilidad, la de crear y configurar un servidor propio, queda prácticamente descartada dadas las limitaciones temporales del TFG. Desarrollar el *backend* de la aplicación desde cero, aunque suponga un control total del mismo, requiere mucho tiempo y puede traducirse en complicaciones futuras si no se es un experto en el tema.

Por lo tanto, para esta situación en concreto, donde todo el desarrollo del proyecto recae en una sola persona, es mejor hacer uso de servicios de terceros que ayuden a consolidar un *backend* fiable y seguro. Además, de esta manera también se ahorra un tiempo que puede utilizarse, por ejemplo, para pulir y mejorar la aplicación en sí.

Por ambos motivos, es aconsejable utilizar un servicio de computación en la nube, que se podría resumir como el alquiler de servidores de terceros. Dentro del mundo de computación en la nube, se pueden encontrar numerosos servicios denominados como *x-aaS*, lo cual puede ser una terminología un tanto confusa, por lo que se va a explicar qué son exactamente cada uno de estos servicios [16].

Los más famosos son los servicios IaaS[15], PaaS[16] y BaaS[17]. IaaS es el servicio más simple, pues ofrece a sus usuarios servidores virtuales que ellos tienen que manejar. PaaS, por otro lado, da

[15] *Infrastructure as a Service*.
[16] *Platform as a Service*.
[17] *Backend as a Service*.

a los desarrolladores algunas herramientas para controlar su aplicación (CDNs[18], fácil despliegue...), aunque estos siguen teniendo que encargarse de crear y configurar la base de datos y la lógica del negocio.

Por último, los servicios BaaS ofrecen APIs y varias herramientas para que los diferentes lenguajes de programación se integren con su *backend*. Por otro lado, también proporcionan otros servicios como almacenamiento, análisis y notificaciones *push*, entre otros.

En la Figura 9 se comparan algunas características de estos servicios:

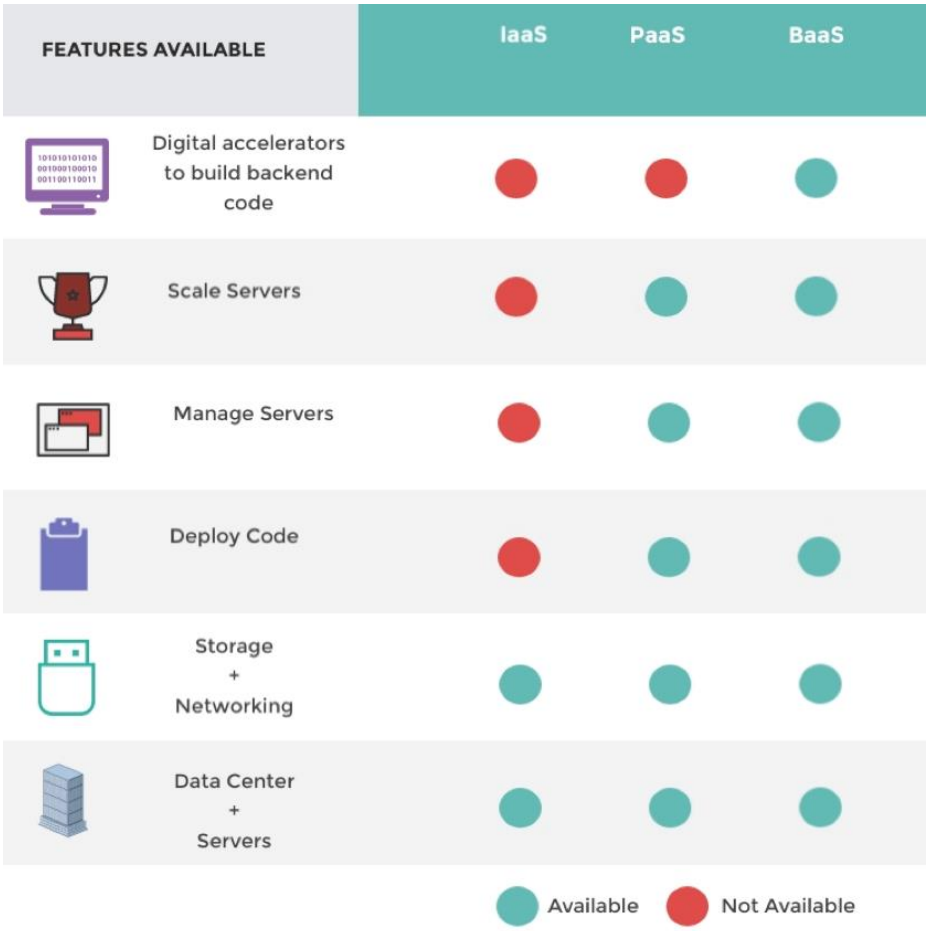


Figura 9. Comparativa entre IaaS, PaaS y BaaS.
(Fuente: https://blog.back4app.com/wp-content/uploads/2019/07/iaasxpaasxbaas.png)

[18] Servicio que permite servir contenidos estáticos (imágenes, PDF, vídeos, CSS, JS...) desde servidores que están más cercanos geográficamente al visitante que entra a la web.

Dentro de cada uno de estos tipos, existe un amplio abanico de servicios para desarrollar el *backend*. Para ir filtrando estas opciones, se han descartado aquellos servicios que son de pago o que requieren una tarjeta de crédito para comenzar con su uso.

A continuación, se han tenido en cuenta las dos características clave de la *app*: la sincronización de datos y la subida de archivos de gran tamaño. Por ambos motivos, se ha buscado un servicio que asegurase la sincronización en tiempo real y que proporcionase una gran capacidad de almacenamiento.

En definitiva, el servicio que finalmente se utilizará para el *backend* de la *app* va a ser *Firebase* [17]. Esta plataforma BaaS fue comprada por *Google* en 2014 y tiene el objetivo de permitir el desarrollo fácil y rápido de aplicaciones móviles de calidad.

Para ello, *Firebase* ofrece una gran variedad de servicios, como almacenamiento de datos en tiempo real, almacenamiento de archivos, autenticación de usuarios y monitorización del rendimiento, entre otros.

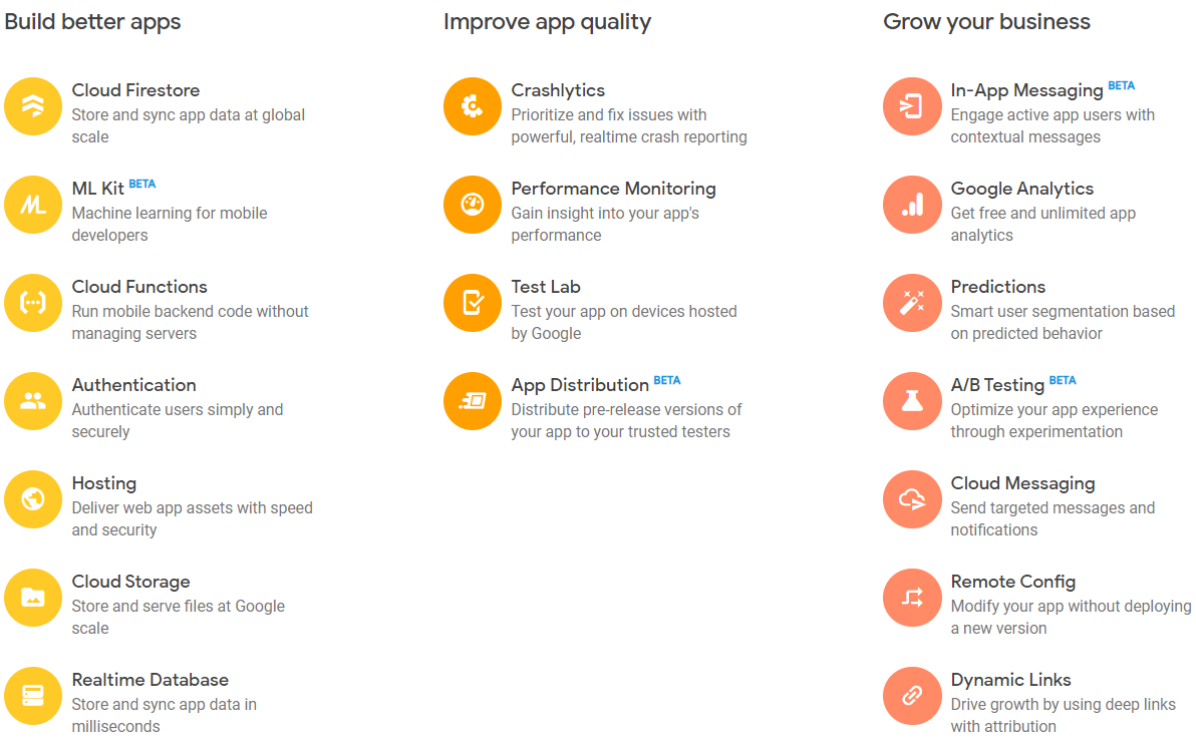


Figura 10. Servicios ofrecidos por Firebase.
(Fuente: https://firebase.google.com/)

En cuanto al precio, *Firebase* ofrece tres tipos de planes, acordes a las necesidades de los distintos clientes. A pesar de ser el más limitado, el plan gratuito "*Spark*" proporciona servicios y herramientas más que suficientes para la *app* en cuestión.

Por último, otro de los principales motivos por los que se ha elegido *Firebase* es su extensa documentación. Ya sea mediante su propia web o a través de los vídeo-tutoriales de su canal de *Youtube*, *Firebase* pone a disposición de los desarrolladores una documentación muy diversa y de alta calidad.

4.4. APIs a utilizar

4.4.1. Mapas

Uno de los requisitos pensados para la aplicación es la posibilidad de introducir los lugares a visitar, mostrando su ubicación en un mapa. Para ello, se hará uso de una API gratuita que permita la visualización y búsqueda de localizaciones.

Se han comparado las distintas alternativas actuales en el mercado y se ha optado por el uso de *Mapbox* [18]. Este servicio ofrece varias APIs interesantes para la aplicación, como la de mostrar mapas y la de hacer búsquedas. *Mapbox* ofrece una opción gratuita para *apps* que no tengan mucho tráfico de usuarios. Además, cabe destacar que permite personalizar de una forma muy rápida los mapas a utilizar.

4.4.2. Previsión meteorológica

Esta es otra funcionalidad que podría implementarse para la *app*. La ventaja sería que los usuarios podrían conocer qué tiempo haría durante las fechas en las que viajan en los lugares a visitar. Sería de gran utilidad a la hora de decidir, por ejemplo, qué llevarse en la maleta.

Para ello, se utilizará la API de *OpenWeatherMap* [19]. Con el plan gratuito se puede obtener el tiempo previsto para los próximos 5 días, siendo necesario especificar la ciudad (mediante ID, nombre o coordenadas).

5. Objetivos

Definir cuáles son el objetivo principal y los subobjetivos del TFG es de gran utilidad para establecer metas concretas para el proyecto. Esto también puede utilizarse como guía durante el desarrollo del Trabajo de Fin de Grado.

A la hora de establecer los objetivos, se ha decidido seguir el principio SMART[19], propuesto por Doran [20]. En resumen, este principio aboga por concretar los objetivos de manera que resulten precisos y claros.

El principal objetivo del proyecto es la realización de una aplicación móvil que permita planear y definir los distintos aspectos de un viaje de forma colaborativa, mediante una interfaz muy cuidada e intuitiva. A partir de esta meta principal, se pueden fijar varios subobjetivos que también se pretenden cumplir:

- Que los datos que se comparten entre los distintos miembros del viaje estén sincronizados en tiempo real.
- Publicar la aplicación desarrollada en la *Play Store* para que cualquier persona con un *smartphone* con *Android* pueda descargarla y utilizarla.
- Que dicha aplicación no ocupe mucho espacio, que sea ligera. Siempre es un punto en contra para los usuarios tener que descargar y utilizar una *app* que ocupe mucha memoria en sus dispositivos.
- Que los usuarios que hagan uso de la aplicación estén, en mayor o en menor medida, satisfechos con la misma. Para ello, se revisarán y se tendrá muy en cuenta las valoraciones y las críticas que estos usuarios dejen patentes en la página de la *app*, en la *Play Store*.

Por otro lado, también merece la pena fijarse otro tipo de subobjetivos, más abstractos y personales, pero que son igual de útiles, pues se traducen en metas a alcanzar durante el desarrollo del proyecto:

[19] Siglas de SMART: *Specific, Measurable, Attainable, Realistic, Timely*.

- Aprender a trabajar con un nuevo *stack*[20] tecnológico y a desarrollar una aplicación para *smartphones*.
- Aprender y coger soltura trabajando otros aspectos de los proyectos multimedia distintos a la programación. Por ejemplo, desarrollar más la faceta de diseño gráfico y diseño de UX[21]/UI[22].
- Crear una herramienta que solvente el problema original de este TFG: la mala experiencia al haber organizado un viaje con amigos. El objetivo es poder utilizar en viajes futuros la aplicación resultante y que realmente facilite el proceso.
- Haber implementado desde cero una aplicación de calidad que me represente como desarrolladora y que pueda mostrar en un *portfolio*[23] personal.

[20] Lista de todos los servicios tecnológicos utilizados para construir y ejecutar una sola aplicación.
[21] *User eXperience*.
[22] *User Interface*.
[23] Manera de presentar trabajos personales de manera organizada para que futuros empleadores y colaboradores puedan verlos rápidamente.

6. Metodología

En este apartado se habla de cómo se va a trabajar durante todo el proyecto. Para ello, se van a especificar las metodologías que se van a emplear y los motivos de su elección, así como las herramientas a utilizar relacionadas con este tema.

Al ser un proyecto individual, es más fácil modificar o combinar varias metodologías. Para lo relacionado con la redacción de la memoria del TFG, se ha optado por seguir una metodología *Kanban*. Respecto a la implementación del proyecto, es decir, toda la parte relacionada con el desarrollo y despliegue de la *app* en sí, se utilizará algo similar a *Scrum*. Esto se debe a que no se trabaja en equipo (así que no hay roles ni reuniones), pero sí se adoptarán conceptos propios de *Scrum* como el *sprint* y *backlog*[24].

Ambas metodologías ágiles planifican el trabajo en torno a un panel de tareas o tablero. En este, se van clasificando las tareas según su estado. Por ejemplo, la práctica habitual es tener dividir las tareas en columnas como "Por hacer", "En progreso", "Pendientes" y "Terminadas".

Estas metodologías sirven para indicar qué, cuándo y cuánto producir. De esta manera, evitan que un equipo de trabajo o un desarrollador acabe produciendo de más. Además, permite ver las tareas de una forma muy visual y aporta más flexibilidad para enfrentarse a cambios.

No obstante, existen numerosas diferencias entre ambas. La principal es que *Kanban* es una metodología en la que hay un trabajo continuo, sin iteraciones, mientras que en *Scrum* sí hay ciclos fijos (conocidos como *sprints*) [21].

Durante la redacción de la memoria, un proceso estático y sin muchos cambios, es apropiado utilizar *Kanban*. Por otro lado, al implementar la *app*, sería aconsejable definir iteraciones o *sprints* para ir mejorando poco a poco el producto final. Como este desarrollo es más dinámico y se puede dividir en numerosas tareas, es adecuado utilizar la metodología *Scrum*.

Para poner en práctica tanto *Kanban* como *Scrum*, se está empleando la herramienta *Trello* [22], un sitio web para gestionar proyectos. Se han creado varias columnas que denotan el estado de una tarea y estas se moverán de unas a otras según se avance en ellas.

[24] Lista de características que han sido priorizadas.

En la siguiente figura, en "*To Do*" aparecen las tareas a realizar. En "*In progress*", se colocan aquellas sobre las que se está trabajando actualmente. "*Pending*" hace referencia a las tareas que necesitan el visto bueno por parte del tutor y "*Done*" recoge las finalizadas.

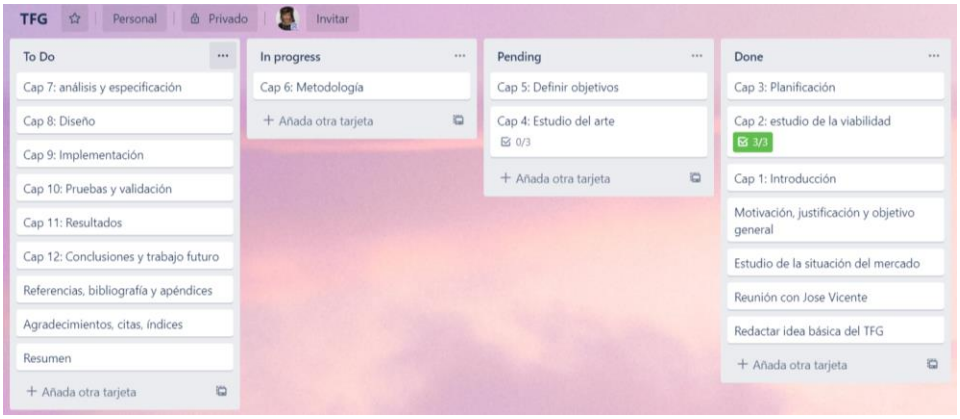


Figura 11. Estado actual del tablero Kanban en Trello.
(Fuente propia)

Cuando se pase a la implementación de la aplicación, se utilizará un tablero *Scrum*. Se cambiará "*To Do*" por "*Backlog*" y se añadirá la columna "*Current sprint*". En esta se colocarán las tareas que se planean realizar durante un *sprint*, que durará aproximadamente una semana.

Otra de las herramientas a utilizar es *Toggl* [23]. Esta web permite contabilizar el tiempo que se le dedica a un proyecto y a sus tareas. Contar las horas sirve, por ejemplo, para comprobar si se está dedicando demasiado tiempo a una tarea o para averiguar si se ha sobreestimado o infraestimado algún apartado del TFG.

Figura 12. Tiempo dedicado al TFG en el mes de noviembre.
(Fuente propia)

7. Análisis y especificación

En este apartado, se recoge la información que servirá de guía para diseñar la solución al problema de la gestión de viajes en grupo. En concreto, se va a utilizar la herramienta de análisis y especificación IEEE 830 [24]. Seguir este estándar ayuda a definir aquellos aspectos que se deben tener en cuenta a la hora de diseñar la aplicación.

Como resultado, en esta sección se han analizado y detallado los distintos tipos de usuarios de la *app* y cuáles son los requisitos que el sistema debe proporcionar y cumplir.

7.1. Tipos de usuarios

En primer lugar, la aplicación diferencia entre 3 tipos de usuarios según los privilegios y el uso que cada uno de ellos hará de la *app*:

Tabla 4. Tipos de usuarios.

Tipo	Descripción
Usuario	Usuario básico de la aplicación. Podrá crear viajes y unirse a otros si un usuario amigo lo añade. En el segundo caso, podrá gestionar prácticamente toda la información de un viaje, como el itinerario, presupuesto y el *checklist*... No podrá gestionar los participantes.
Administrador del viaje	Se trata del usuario creador de un viaje o, si este ha abandonado el viaje, es la persona que lleva más tiempo como participante. Puede hacer lo mismo que un usuario normal, pero también tiene la capacidad de gestionar los participantes de un viaje. Podrá añadir nuevos usuarios siempre y cuando los tenga agregados como amigos.
Administrador	Administrador de la aplicación. Sus funciones principales son encargarse de toda la gestión de usuarios y de asegurar el correcto funcionamiento de la *app*.

7.2. Requisitos

A continuación, se recogen cada uno de los requisitos del proyecto. Estos definirán las funcionalidades de la aplicación. Los requisitos disponen de un identificador (RF para los requisitos funcionales y RNF para los no funcionales), el tipo de usuario con el que está relacionado y una corta descripción.

Además, en el campo "Tipo" se diferencia entre requisitos que se han de implementar obligatoriamente (básico) y requisitos opcionales (avanzado), que se desarrollarán si se dispone de tiempo suficiente.

7.2.1. Requisitos funcionales

Son aquellos relacionados con las funcionalidades de la aplicación. Describen las actividades y servicios que la aplicación deberá proveer.

Tabla 5. Requisitos funcionales.

Identificador	RF1
Usuario	Usuario, administrador del viaje
Tipo	Básico
Descripción	Gestionar viajes (crear nuevos, dejar de pertenecer a un viaje…)

Identificador	RF2
Usuario	Usuario, administrador del viaje
Tipo	Básico
Descripción	Posibilidad de crear un itinerario de viaje con los lugares que se van a visitar, pudiendo indicar otros datos como la hora, ubicación, el tipo de plan, precio, etc.

Identificador	RF3
Usuario	Usuario, administrador del viaje
Tipo	Básico

Descripción	Llevar un control del presupuesto de un viaje. Poder visualizar el dinero gastado y en qué se ha gastado.

Identificador	RF4
Usuario	Usuario, administrador del viaje
Tipo	Básico
Descripción	Compartir fotografías, vídeos, documentos y otros archivos con el resto de participantes del viaje.

Identificador	RF5
Usuario	Usuario, administrador del viaje
Tipo	Básico
Descripción	Posibilidad de hacer una lista de objetos y pertenencias que se desean llevar al viaje.

Identificador	RF6
Usuario	Usuario, administrador del viaje
Tipo	Básico
Descripción	Redactar, como si se tratase de un diario, las memorias y experiencias vividas durante cada día, y compartirlas con el resto de participantes del viaje.

Identificador	RF7
Usuario	Usuario, administrador del viaje
Tipo	Básico
Descripción	Gestionar los usuarios que son amigos.

Identificador	RF8
Usuario	Usuario, administrador del viaje
Tipo	Básico
Descripción	Gestionar los usuarios bloqueados.

Identificador	RF9
Usuario	Usuario, administrador del viaje
Tipo	Básico
Descripción	Gestionar información básica del propio perfil de usuario (e-mail, *nickname*[25], foto de perfil…).

Identificador	RF10
Usuario	Administrador del viaje
Tipo	Básico
Descripción	Gestión de los usuarios que participan en el viaje.

Identificador	RF11
Usuario	Usuario, administrador del viaje
Tipo	Avanzado
Descripción	Poder planificar un nuevo viaje tomando como base una plantilla ya creada, la cual ha sido publicada por otro usuario. De la misma manera, tener la posibilidad de publicar la plantilla de un viaje propio tras haberlo finalizado.

Identificador	RF12
Usuario	Usuario, administrador del viaje
Tipo	Avanzado
Descripción	Conocer la previsión meteorológica del lugar al que se viaja en las fechas indicadas.

Identificador	RF13
Usuario	Administrador
Tipo	Básico
Descripción	Gestión de usuarios de la *app*.

[25] Apodo. Nombre de usuario.

Identificador	RF14
Usuario	Administrador
Tipo	Básico
Descripción	Visualización de gráficos y otro tipo de indicadores para poder comprobar si la aplicación funciona correctamente.

7.2.2. Requisitos no funcionales

Los requisitos no funcionales hacen referencia al comportamiento del propio sistema, a la vez que señalan las restricciones del mismo. El campo "Tipo", en esta ocasión, indica el área con el que están relacionados.

Tabla 6. Requisitos no funcionales.

Identificador	RNF1
Usuario	Sistema
Tipo	Usabilidad
Descripción	La aplicación dispondrá de una interfaz intuitiva, que pueda ser utilizada por cualquier persona acostumbrada a manejar *apps*, sin necesidad de pasar por un proceso de aprendizaje largo.

Identificador	RNF2
Usuario	Sistema
Tipo	Accesibilidad
Descripción	El diseño de la *app* cuidará los aspectos relacionados con la accesibilidad, tales como el contraste de colores, la legibilidad y el número de *clicks* necesarios para realizar una acción.

Identificador	RNF3
Usuario	Sistema
Tipo	Disponibilidad

Descripción	La aplicación estará disponible de forma permanente para todos los usuarios que la hayan descargado. Esta podrá descargarse a través de la *Play Store*.

Identificador	RNF4
Usuario	Sistema
Tipo	Eficiencia
Descripción	El tiempo de respuesta del sistema deberá ser rápido. No deberá tardar más de 3 segundos en responder a las acciones que realice el usuario.

Identificador	RNF5
Usuario	Sistema
Tipo	Confidencialidad
Descripción	El acceso a los datos de los usuarios y de los viajes se restringirá únicamente a los administradores del sistema y a los usuarios en cuestión.

Identificador	RNF6
Usuario	Sistema
Tipo	Legal
Descripción	Se contará con las licencias de todas herramientas y materiales empleados en la construcción de la *app*.

8. Diseño

En este capítulo, que es uno de los más importantes y extensos del Trabajo de Fin de Grado, es donde se diseña la solución a implementar.

Especificar todos los aspectos relacionados con el diseño de la solución facilita la etapa de implementación, pues se evita tener que tomar decisiones sobre la marcha. Así se ahorra tiempo y se aseguran mejores resultados. Este apartado se puede considerar una guía a seguir cuando se pase a la fase del desarrollo de la aplicación.

8.1. Diseño de la arquitectura conceptual

La arquitectura conceptual permite hacerse una idea de cuál va a ser la estructura de la aplicación a grandes rasgos, sin entrar en tecnologías concretas. Se podría decir que es la arquitectura base de la solución a desarrollar.

Además, tras realizar el diseño de la arquitectura conceptual, resulta mucho más fácil distinguir los bloques que van a conformar la solución. En la siguiente figura se puede ver cuál es la estructura de la aplicación móvil:

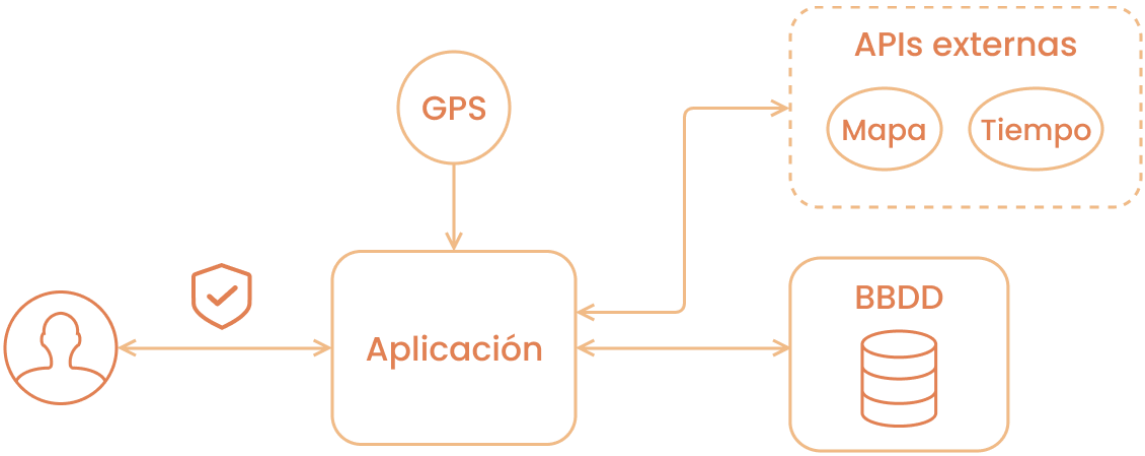


Figura 13. Diagrama de la arquitectura conceptual.
(Fuente propia)

En primer lugar, el usuario se comunicará directamente con la *app*. Para ello, este tendrá que haber iniciado sesión previamente. La lógica de la aplicación será la encargada de hacer las peticiones a la base de datos.

Por otro lado, la aplicación se nutrirá de APIs externas para obtener previsiones meteorológicas [RF12] y mostrar mapas de utilidad a la hora de crear los itinerarios. Además, se utilizará la información que proporciona el GPS del móvil para conocer la ubicación de los usuarios.

8.2. Diseño de la arquitectura tecnológica (front/back-end)

Para el diseño de la arquitectura tecnológica, se va a coger de base el anterior diagrama de la arquitectura conceptual. En esta sección es el momento de concretar qué tecnologías se van a usar en cada bloque.

En el apartado del estudio del arte, se investigó cuáles eran las tecnologías más óptimas para solucionar el problema de organizar viajes en grupo. Teniendo en cuenta que la solución es una aplicación para móviles, el diagrama de su *stack* tecnológico es el siguiente:

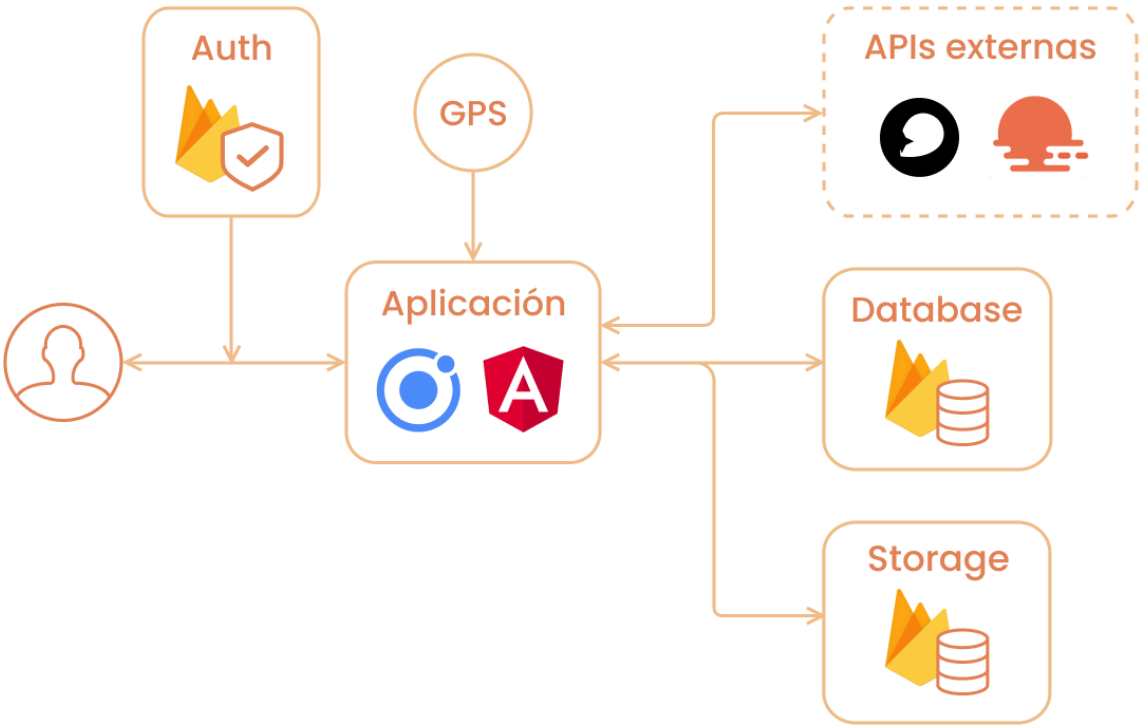


Figura 14. Diagrama de la arquitectura tecnológica.
(Fuente propia)

Para implementar la aplicación, se va a utilizar el *framework Ionic* basado en *Angular*. En cuanto al registro y la autenticación se empleará el SDK[26] de *Firebase*, para conseguir que se haga de forma segura. La aplicación se comunicará las APIs externas de *Mapbox* (para visualizar mapas) y *OpenWeatherMap* (para obtener previsiones meteorológicas).

Mientras tanto, la lógica de la aplicación se servirá de librerías y otros SDK de *Firebase* para acceder a sus bases de datos. En concreto, esta plataforma para el desarrollo de aplicaciones proporciona distintas bases de datos, las cuales se adaptan a unas necesidades u otras.

Por un lado, está la llamada "*Database*", que es NoSQL, donde se va a almacenar la información de un usuario y de los viajes. Dentro de esta se ha de elegir entre "*Cloud Firestore*" y "*Realtime Database*". Se ha optado por la base de datos "*Cloud Firestore*", ya que es la que recomienda la propia plataforma, al ser una versión más nueva y optimizada de la segunda.

Por otra parte, *Firebase* tiene otra base de datos, "*Storage*", más enfocada al almacenamiento de archivos pesados tales como fotos, vídeos, documentos, etc. Será en "*Storage*" donde se guardarán los archivos que los usuarios suban para compartirlos con el resto de participantes de los viajes.

8.3. Diseño de la persistencia

8.3.1. Modelo de datos

Para el almacenamiento de la información, como se ha comentado en el diseño de la arquitectura tecnológica, se va a hacer uso de las bases de datos de *Firebase*.

Antes de comenzar a diseñar el modelo, se debe conocer a fondo cómo son y de qué manera funcionan las bases de datos de *Firebase*. En concreto, a partir de ahora se va a profundizar acerca de la "*Cloud Firestore*", que es la elegida al ser una versión mejorada de la original "*Realtime Database*".

[26] *Software Development Kit* (kit de desarrollo de *software*).

La "*Cloud Firestore*" es NoSQL, es decir, es una base de datos no relacional con un modelo de datos propio. Las principales ventajas de las bases de datos NoSQL son la libertad a la hora de almacenar los datos y la velocidad de lectura.

Es cierto que, a la hora de modificar un dato ya existente, para las NoSQL es un proceso más costoso, pues en este tipo de bases de datos la misma información no solo se almacena en un sitio concreto: se tienen que buscar todos los lugares donde se encuentra ese dato y cambiarlo.

No obstante, la filosofía de estos modelos no relacionales es darles prioridad a las lecturas por encima de las escrituras. Al fin y al cabo, las lecturas se realizan con mucha más frecuencia. De esta forma se conseguirá que la *app* sea más rápida [RNF4].

La *Cloud Firestore* guarda los datos en pares de clave-valor dentro de los denominados documentos, que son la unidad de almacenamiento. Estos documentos, a su vez, se almacenan en colecciones, unos contenedores que ayudan a agrupar y a organizar los documentos.

Figura 15. Elementos de la Cloud Firestore.
(Fuente: https://firebase.google.com/docs/firestore?hl=es-419)

La raíz de la *Cloud Firestore* tiene colecciones (no puede haber documentos sueltos en la raíz) y estas, documentos. Luego, los documentos podrán tener subcolecciones, pero no otros documentos. Las subcolecciones son colecciones asociadas a un documento específico. En resumen, el patrón que sigue la estructura de la *Cloud Firestore* es: colección > documento > subcolección > documento > subcolección... y así las veces que se deseen.

Un buen comienzo para definir el modelo de datos es pensar, a grandes rasgos, qué objetos va a tener la aplicación. En este caso, la *app* tendrá viajes, usuarios, itinerarios, *checklists*, carpetas con archivos, etc. El siguiente paso consistiría en definir cómo se relacionan entre sí estos objetos y qué campos tiene cada uno.

Después de este primer análisis, hay que examinar cómo mejorar la organización de estos datos, de forma que se haga CRUD[27] a la *Cloud Firestore* de la forma más óptima posible. Por eso, hay que tener claros ciertos conceptos que influyen en el diseño del modelo de datos y también es vital tener una idea general de cómo se va a usar la *app*.

El primero de ellos es que no se puede obtener información parcial de un documento. Esto quiere decir que no se podría obtener solo dos campos de un documento, sino que se tendría que descargar toda la información de este. Por lo tanto, según cómo se vaya a utilizar cierta información dentro de la *app*, será más recomendable mantenerla junta o separada.

Otro dato relevante de la *Cloud Firestore* es que cobra por el número de peticiones a la base de datos. Por este motivo, hay que estructurar la información de forma que no se realicen muchas llamadas. Lo más probable es que no se supere el límite gratuito, pero sigue siendo un factor a tener en cuenta.

Conociendo todas estas reglas, se puede comenzar con el diseño del modelo de datos. En la Figura 16 se muestra el resultado final:

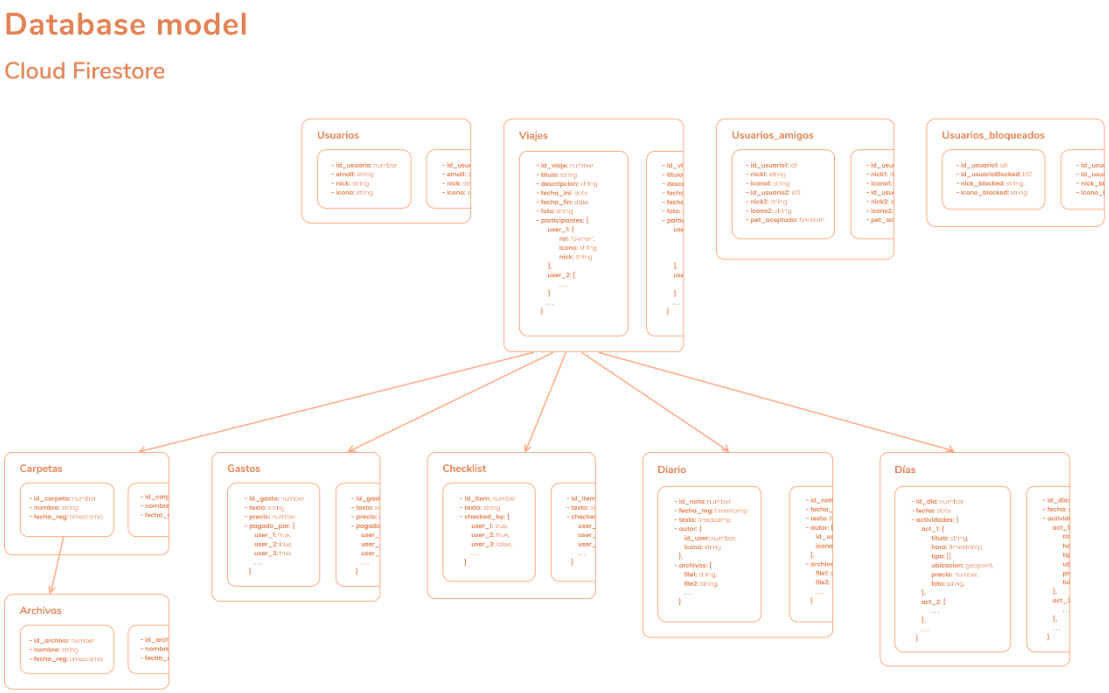


Figura 16. Diseño del modelo de datos.
(Fuente propia)

[27] CRUD: *Create, Read, Update* y *Delete*. Son las funciones principales para gestionar una base de datos.

Ahora se va a analizar los distintos componentes que forman el diagrama, puesto que a grandes rasgos no se puede apreciar su contenido.

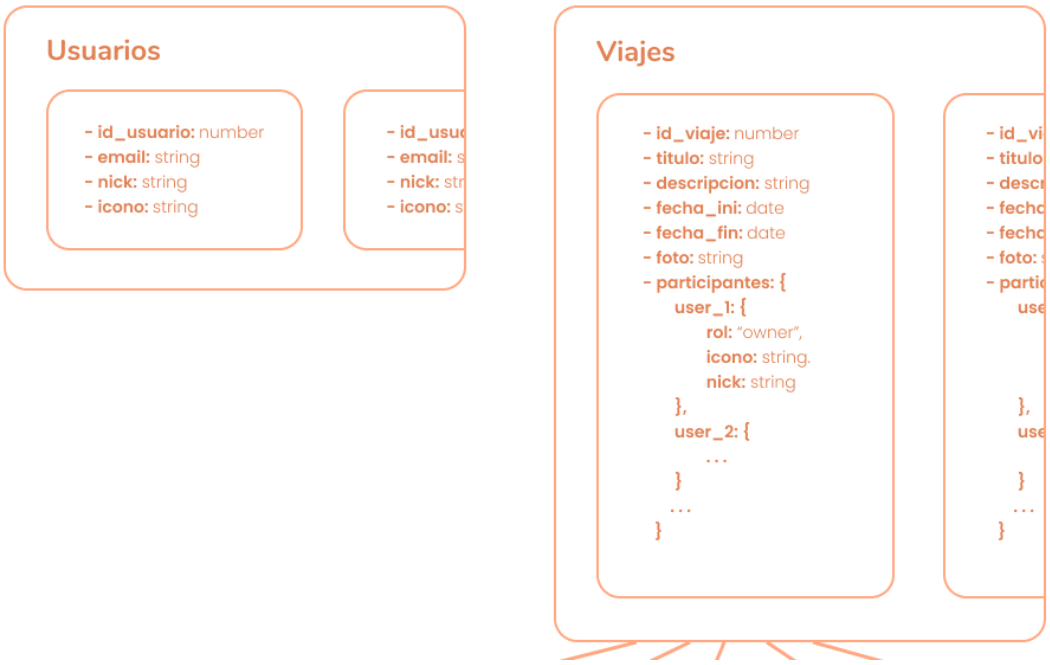


Figura 17. Detalles del modelo de datos (I).
(Fuente propia)

En primer lugar, hay una colección de usuarios, donde cada documento tendrá la información básica de cada uno de ellos [RF9]. Después están los viajes, el elemento principal de la aplicación, donde se almacenan datos como el título, fecha de inicio, etc. [RF1]

Los documentos de viajes también tienen un mapa de participantes en el que se almacena, para cada uno de estos IDs de usuario, el icono del usuario y su rol dentro del viaje. Este rol podrá ser de editor o de propietario, puesto que será el propietario quien tenga más privilegios (gestionar los participantes del grupo [RF10]).

Si bien es cierto que se podría haber sacado una subcolección para almacenar los participantes (lo cual aportaría una mayor privacidad), habría supuesto un problema a la hora de listar los viajes que tiene un usuario [RF1]. A día de hoy, la *Cloud Firestore* no habría permitido obtener todos los viajes accediendo a información de una subcolección.

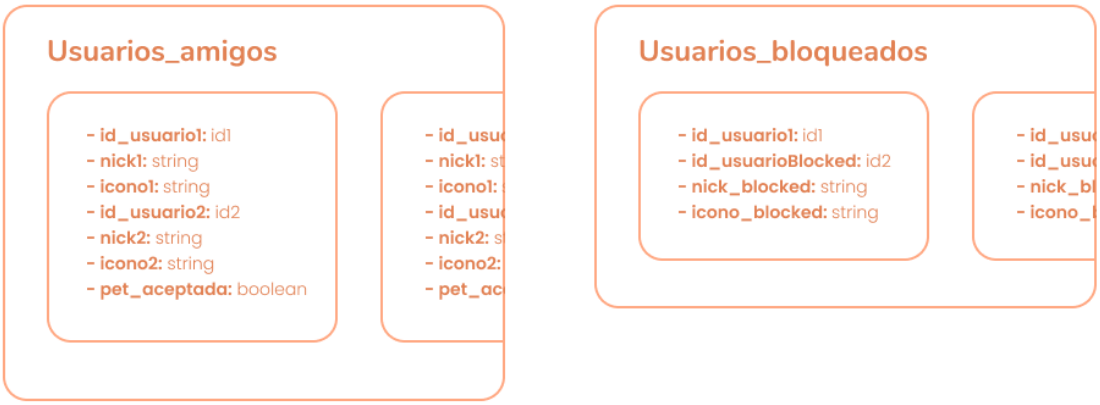


Figura 18. Detalles del modelo de datos (II).
(Fuente propia)

Por otro lado, se encuentran las colecciones encargadas de almacenar los amigos [RF7] y las personas a las que ha bloqueado un usuario [RF8]. En concreto, en "Usuarios_amigos" se guardan los datos básicos de cada uno de ellos (ID, *nick* y foto de perfil) y el estado de la petición de amistad (si son ya amigos o si todavía no se ha aceptado).

Por otra parte, en "Usuarios_bloqueados" solo se almacena la información básica del usuario que ha sido bloqueado. La diferencia respecto a la anterior colección se debe a que, si dos usuarios son amigos, ambos deberían poder ver la información del otro usuario. Sin embargo, si a un usuario lo bloquean, este no debe saber quién lo ha bloqueado, por lo que guardar el *nick*, icono... del usuario que ha bloqueado no es necesario.

Otra alternativa podría ser guardar toda esta información de amigos y bloqueados en una subcolección de cada usuario. Sin embargo, esto no es óptimo y se puede comprobar con un ejemplo. Si un usuario al que se le tiene bloqueado se cambia el *nickname*, se haría la modificación dentro de "Usuarios" y después la *Cloud Firestore* tendría que recorrer las subcolecciones de todos los usuarios, comprobar si lo tienen bloqueado y cambiar el *nick* en esos documentos.

Esto es computacionalmente mucho más costoso que almacenar las relaciones entre usuarios en una colección directamente. En este caso, la *Cloud Firestore* indexaría de forma más fácil y rápida los documentos en los que se encuentra el usuario bloqueado.

A continuación, se hablará en detalle de las subcolecciones que tienen los viajes, de izquierda a derecha:

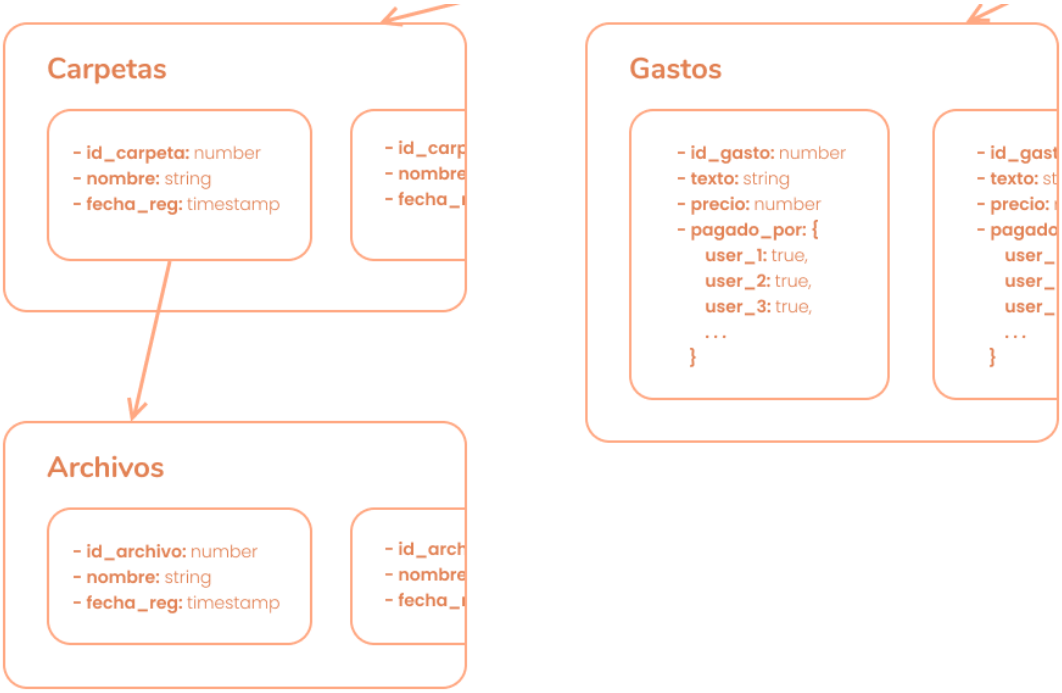


Figura 19. Detalles del modelo de datos (III).
(Fuente propia)

La primera subcolección de los viajes es la llamada “Carpetas”, que se utilizan para almacenar archivos como fotografías, vídeos, etc. [RF4] Por su parte, los archivos contienen, entre otros campos, cuál es su nombre (la ruta donde se encuentra el fichero).

Por otro lado, se tiene la subcolección de gastos. Es la encargada de guardar todas las expensas que se producen en un viaje: cuánto ha costado, de qué tipo, cuándo... [RF3] Dentro de un gasto también se encuentra el mapa “pagado_por”.

Este mapa almacena los usuarios relacionados con el gasto y muestra, para cada uno de ellos, si lo han pagado o no. En el caso de que sea un gasto propio, aparecerá solamente un usuario. Esto supone mucha flexibilidad, pues permite crear gastos individuales y gastos en grupo.

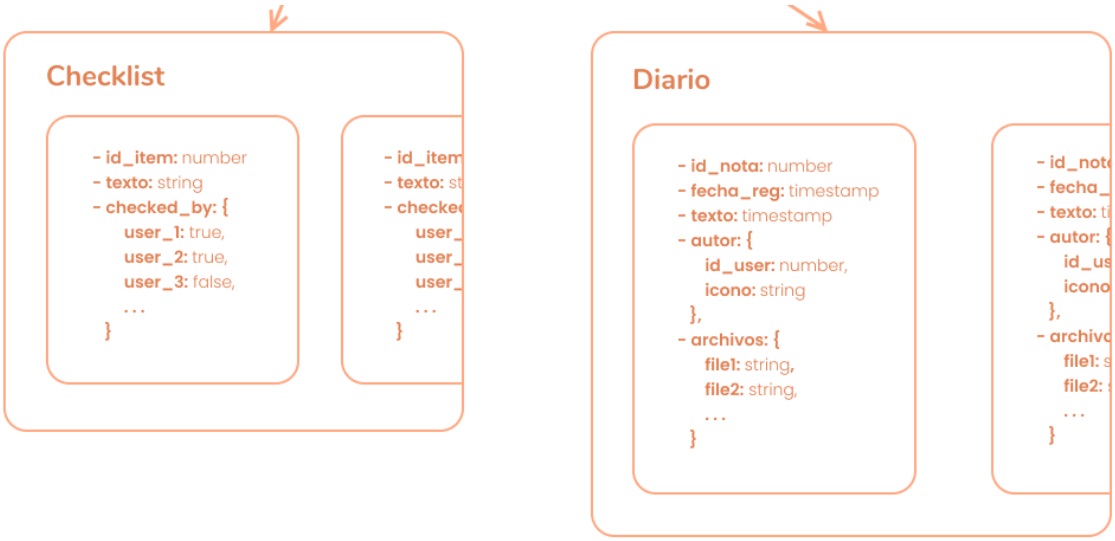


Figura 20. Detalles del modelo de datos (IV).
(Fuente propia)

La siguiente subcolección que se encuentra es la de *checklist*. Este *checklist*, en principio, tiene la utilidad de apuntar objetos necesarios para llevarse de viaje [RF5], aunque también se puede utilizar como un *To Do list*[28]. Por ejemplo, se podría apuntar que es necesario haberse vacunado de una enfermedad común del país a visitar.

En "C*hecklist*" se guarda en qué consiste el ítem y quiénes lo han marcado como hecho. En caso de que se quiera apuntar algo en el *checklist* de forma privada, el mapa de "*checked_by*" solo guardará el ID de ese usuario.

En cuanto al diario, servirá para redactar anécdotas y experiencias del viaje [RF6]. Lo más destacable de sus documentos es que almacenan quién ha escrito la nota y también que permite adjuntar archivos a esta.

[28] Lista de cosas pendientes por hacer.

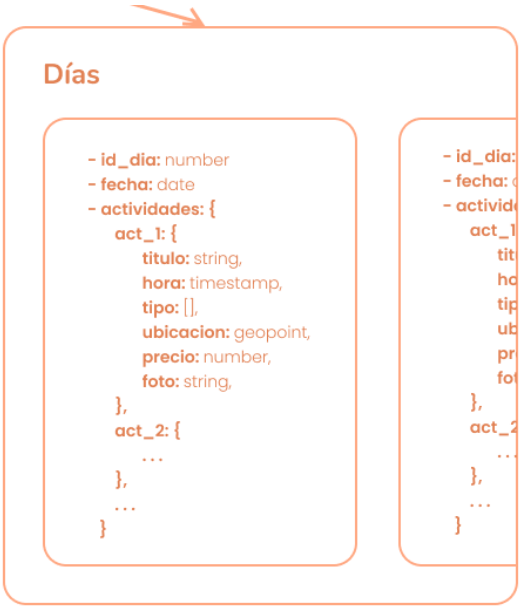


Figura 21. Detalles del modelo de datos (V).
(Fuente propia)

Por último, en la figura anterior aparece la subcolección más importante, pues es la encargada de almacenar los planes de cada día [RF2]. Cabe destacar el mapa de actividades, que guarda todo lo que se tiene pensado realizar en un día, incluyendo el tipo, el precio, una foto, etc.

8.3.2. Gestión de contenidos

Antes de continuar hablando del diseño del proyecto, hay que tratar el tema relacionado con la gestión de contenidos. Esto hace referencia a todo lo relacionado con los permisos: quién puede añadir usuarios a un viaje, si se puede borrar algo que ha subido otro usuario, etc.

Para entender cómo funciona la gestión de contenidos, se tiene que tener en cuenta una precondición muy importante: se trata de un entorno de confianza.

El objetivo del proyecto es permitir la organización de viajes, tanto individuales como grupales. En el caso de los viajes en grupo, prácticamente la totalidad de ellos estarán organizados por grupos de amigos, familiares o similar.

Por lo tanto, se supone que los usuarios con los que se participa en los viajes son personas de un entorno cercano, personas con las que se tiene confianza. Este hecho condiciona prácticamente todo lo relacionado con los permisos dentro de la aplicación.

En primer lugar, dentro de un viaje habrá un único administrador (aunque en un futuro se podría contemplar la posibilidad de permitir más de uno). Este usuario, comparado con los demás, solo tendrá un privilegio extra: la gestión de participantes de un viaje. El resto de usuarios no podrán ni añadir ni quitar participantes. Si el administrador se sale del grupo del viaje, pasará a serlo el siguiente usuario con más antigüedad.

La mayoría de contenido de la *app* podrá ser gestionado por igual por cualquier usuario. En concreto, estos son los datos de un viaje, las actividades del itinerario, los ítems del *checklist*, los gastos y las carpetas y archivos. Por otro lado, en cuanto a las notas de diario y los gastos y *checklist* privados, solo el usuario creador podrá gestionar su contenido.

Como se puede observar, al tener en cuenta la precondición del entorno de confianza, lo más lógico es no establecer demasiadas medidas de seguridad, puesto que repercutirían negativamente en la experiencia de usuario (por ejemplo, los usuarios se verían obligados a pedirle al administrador que realizase la mayoría de tareas).

En definitiva, los usuarios tendrán mucha libertad con la gestión de un viaje y su contenido. Solo habrá limitaciones respecto a la gestión de usuarios, las notas de diario y demás contenido privado.

8.3.3. Seguridad e integridad

Otro aspecto de la persistencia de datos es su seguridad. Se ha de controlar quién puede acceder a la base de datos y cuáles son sus privilegios [RNF5]. En *Cloud Firestore*, cada tipo de documento tiene sus propias normas de seguridad y se pueden especificar las reglas que se crean necesarias.

Por ejemplo, se podrían crear reglas para evitar que un usuario modifique la nota de otro. En primer lugar, se comprobaría siempre que el usuario esté autenticado. Después, se miraría si ese usuario pertenece a dicho viaje. Si es el caso, se comprobaría si es del autor de la nota. En

este ejemplo, se le denegaría esa modificación. Como es probable que estas comprobaciones se reutilicen varias veces, la *Cloud Firestore* permite crear funciones dentro de las reglas.

La siguiente imagen de la documentación de *Firebase* es un ejemplo de reglas de un documento:

```
service cloud.firestore {
  match /databases/{database}/documents {
    // True if the user is signed in or the requested data is 'public'
    function signedInOrPublic() {
      return request.auth.uid != null || resource.data.visibility == 'public';
    }

    match /cities/{city} {
      allow read, write: if signedInOrPublic();
    }

    match /users/{user} {
      allow read, write: if signedInOrPublic();
    }
  }
}
```

Figura 22. Reglas de seguridad en Cloud Firestore.
(Fuente: https://firebase.google.com/docs/firestore/security/rules-conditions)

La segunda línea de código especifica que las reglas a continuación se aplican a cualquier base de datos existente en la *Cloud Firestore*. Después, hay declarada una función que devuelve verdadero si el usuario está autenticado o si la información está publicada.

Esta función se llama cuando quien envía la petición intenta leer o escribir los datos de las ciudades y de los usuarios. En el caso de la *app*, habría que crear una función que comprobase que el usuario está autenticado, pertenece a ese viaje y tiene los privilegios requeridos.

En cuanto a la integridad de la *app*, se han comparado varias formas de hacer *backups*[29] de la información alojada en la base de datos. En la documentación de *Firebase* hay una guía para exportar los datos. No obstante, es necesario activar la facturación de la *Google Cloud Platform* [25] para llevar a cabo el proceso.

Por este motivo, se utilizará la librería "*node-firestore-import-export*" para hacer copias de seguridad [26]. Esta librería almacena todos los datos en un JSON con la misma estructura, facilitando la posible restauración del contenido de la base de datos.

[29] Copias de seguridad.

8.4. Diseño de Interacción/Experiencia de Usuario

Las sensaciones y emociones que experiencia un usuario al hacer uso de un producto son claves para que este fracase o sea un éxito. Por ejemplo, si la experiencia que tiene un usuario al utilizar una página web es mala, buscará otras alternativas y no volverá a entrar a esta.

Es importante tener en mente que nadie hace uso de un servicio o producto porque les guste utilizarlo, lo hacen para satisfacer sus necesidades y/o problemas. Por ejemplo, ninguna persona usa *WhatsApp* porque le gusta la aplicación en sí, sino porque les permite comunicarse fácilmente y al instante con la gente de su entorno.

Volviendo al ejemplo de las webs, a las personas les da igual qué tecnologías se han empleado, si el sistema que tiene detrás es potente o si se ha invertido mucho dinero en desarrollarla. Lo que les importa es que la web resuelva sus problemas de la forma más óptima posible, y si no están contentos con cómo los resuelven no la utilizarán nunca más.

El objetivo principal de este Trabajo de Fin de Grado es solucionar el problema de organizar viajes en grupo. Para ello y por los motivos antes mencionados, en este apartado se han hecho las investigaciones necesarias para realizar un buen diseño de interacción y de experiencia de usuario. De esta forma, se pretende atraer y retener al mayor número de usuarios posible.

8.4.1. Diseño de Personas

El primer paso para llevar a cabo el diseño de la interacción consiste en definir “Personas”. Una Persona es un personaje ficticio que representa a un grupo de posibles usuarios y, al igual que las personas de la vida real, tienen un nombre, edad, género, aficiones, manías…

Definir Personas con todos estos detalles concretos ayuda a conocer cuáles son las necesidades de estos usuarios y qué es lo que esperan de un producto. Este proceso sirve para pensar de forma diferente, poniéndose en la piel de estas Personas y averiguando sus puntos de vista.

A continuación, se han diseñado 3 Personas con su información básica, ocupación y aficiones, objetivos y necesidades, *pain points*[30] y patrones de comportamiento relevantes, tomando como referencia la entrada del blog de *Qubstudio* [27].

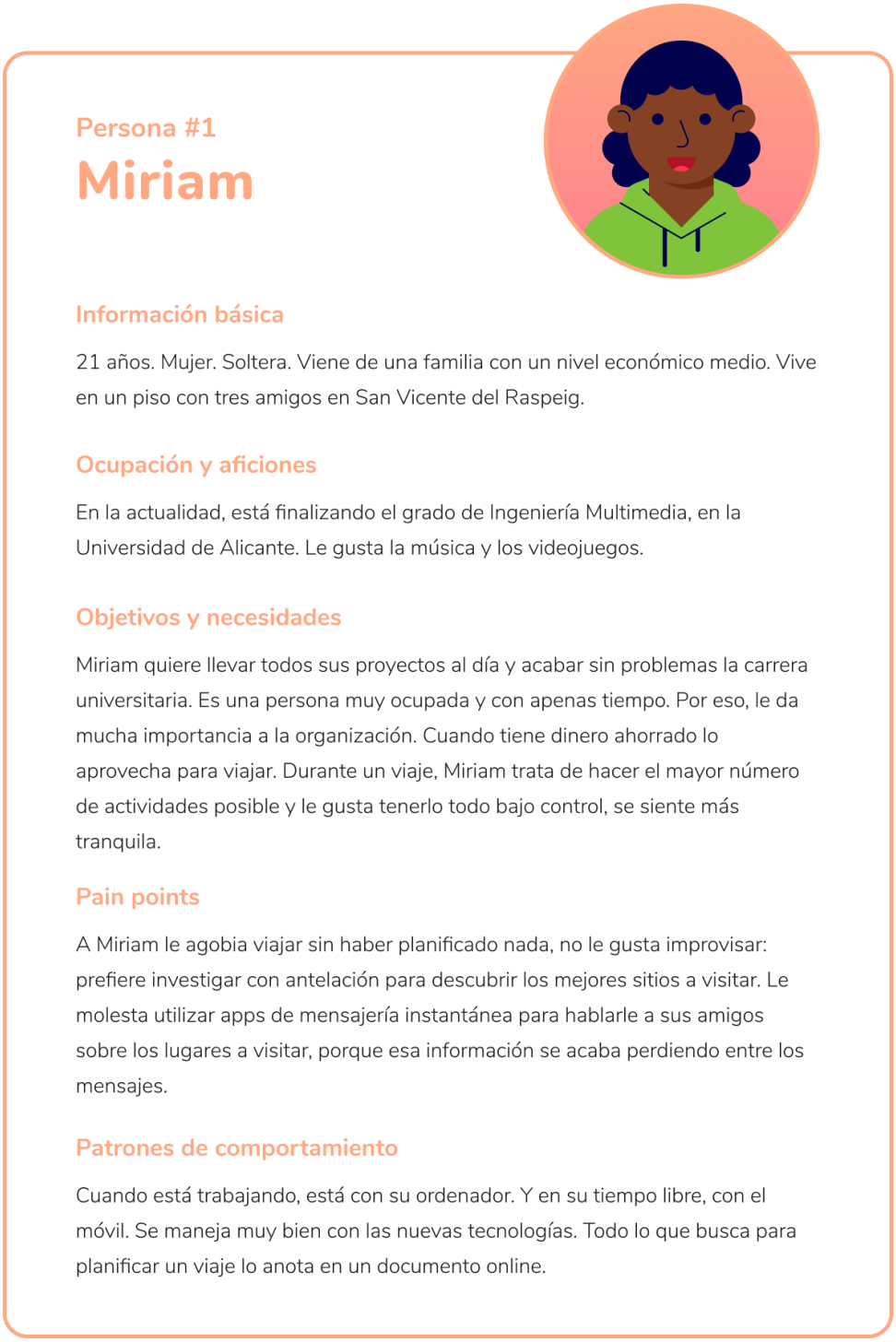
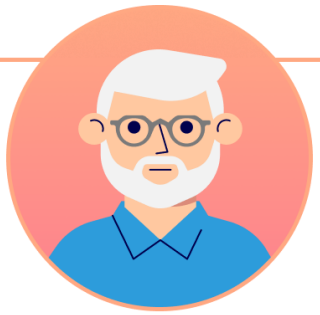


Figura 23. Persona #1.
(Fuente propia)

[30] Preocupaciones o problemas que tiene una persona.

Persona #2

Antonio



Información básica

56 años. Hombre. Casado y con dos hijas. Nivel económico medio con sueldo estable. Vive en un pueblo de la Vega Baja con su mujer.

Ocupación y aficiones

Trabaja desde hace más de 20 años como secretario en un instituto de secundaria. Le apasiona la historia y los cómics.

Objetivos y necesidades

Su día a día consiste en ir al trabajo, volver a casa, dormir la siesta, leer temas relacionados con la historia y cuidar de su madre. Solo tiene vacaciones en el mes de agosto, así que le gusta irse lejos con su mujer y sus amigos para desconectar de la vida cotidiana. Como se lo puede permitir, prefiere que le organicen el viaje y así no tiene que pensar tanto. Los viajes culturales son su tipo favorito de viaje.

Pain points

A Antonio no le gusta viajar sin haberse informado previamente de la historia del lugar al que se dirige. Por otra parte, tiene una cámara de móvil muy buena y está cansado de que su mujer y sus amigos le pidan que les haga fotos constantemente. También le molesta tener que enviarles individualmente esas fotos por WhatsApp.

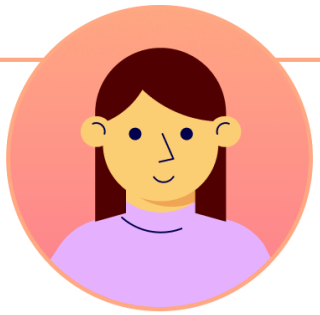
Patrones de comportamiento

Antonio coge el móvil lo menos posible, solo cuando es necesario. No obstante, aunque no sea un experto, se desenvuelve bien con las tecnologías si las usa durante un tiempo y se familiariza.

Figura 24. Persona #2.
(Fuente propia)

Persona #3

Isabelle



Información básica

25 años. Mujer. Soltera. Viene de una familia con un nivel económico medio-bajo. Nació en Francia pero vive en Murcia.

Ocupación y aficiones

Sus padres tienen un bar y ella les ayuda los findes. Estudia el grado de Biología en la universidad. Le gusta ver series y la cocina.

Objetivos y necesidades

Isabelle quiere mejorar su inglés y vivir nuevas experiencias, por eso se irá de Erasmus a Polonia el curso que viene. Quiere ahorrar el máximo dinero posible para el futuro, porque una vez allí le gustaría hacer mini viajes por el centro de Europa. Utilizará mucho las redes sociales para compartir su día a día y le gustaría apuntar todas sus experiencias en una libreta, para poder leerlas pasados los años.

Pain points

Es una persona a la que le cuesta mucho controlarse en cuanto al dinero: nunca lleva efectivo, solo tarjeta, así que nunca sabe exactamente cuánto se gasta. También es algo desorganizada y muy olvidadiza, lo más problable cuando viaja es que se le olvide llevar algo o que pierda alguna de sus pertenencias.

Patrones de comportamiento

Isabelle utiliza mucho su teléfono móvil, se pasa todo el día en redes sociales. Su smartphone, sin embargo, tiene muchos años y le da muchos problemas, por lo que prefiere anotar cosas en papel.

Figura 25. Persona #3.
(Fuente propia)

Como se puede comprobar, Miriam, Antonio e Isabelle tienen vidas y situaciones totalmente distintas entre sí. Lo que sí tienen en común es que de cada uno de ellos se pueden sacar conclusiones valiosas de cara al proyecto.

A Miriam le gusta hacer viajes con sus amigos y tener toda la información sobre los lugares a visitar de forma organizada. Antonio quiere tener un viaje tranquilo sin perder el tiempo enviando a sus amigos las fotos que les hace. Por último, Isabelle necesita anotar sus gastos para no excederse con el dinero y también tener por escrito las pertenencias que lleva consigo.

De estas necesidades, la conclusión que resulta más obvia es la siguiente: sus problemas se podrían solucionar con una aplicación móvil que ofrezca las herramientas adecuadas. ¿Por qué una aplicación móvil? Porque casi todo el mundo posee un *smartphone* propio y porque les permitiría acceder a información, fotos, gastos… de forma inmediata, independientemente de dónde se encuentren.

Hablando en términos generales, esta *app* debería permitir anotar los lugares a visitar de un viaje, los gastos que se van teniendo, los ítems de la maleta y también compartir archivos de forma fácil con tus amigos. Además, la aplicación tiene que ser eficiente y estar optimizada (en beneficio de Isabelle, que tiene un móvil que le da fallos) y ha de ser muy intuitiva (para Antonio, que de primeras le puede resultar complicado usar una *app* que no conoce).

Como se puede observar, esta fase de análisis de Personas también sirve para definir requisitos. En el caso de este TFG, todas las necesidades comentadas en el párrafo anterior ya se han contemplado y añadido en su correspondiente apartado.

8.4.2. User Journey Maps

A continuación, se van a utilizar a las Personas definidas anteriormente para ver cómo reaccionarían ante ciertos casos de uso. Este proceso es lo que se denomina "*User Journey Maps*": visualizar de forma gráfica las fases que sigue un usuario al interactuar con uno de los servicios y anotar la emoción provocada con cada una de ellas [28].

Lo más interesante es repetir el mismo experimento con varias personas para comparar los resultados. Se han hecho varias pruebas utilizando todas las Personas definidas anteriormente

y las diferencias entre sí son insignificantes, por lo que se ha optado por mostrar directamente el *User Journey Map* medio.

El primero de ellos consiste en, dentro de un viaje, añadir usuarios que van a participar en este. Podemos ver el resultado en el siguiente gráfico:

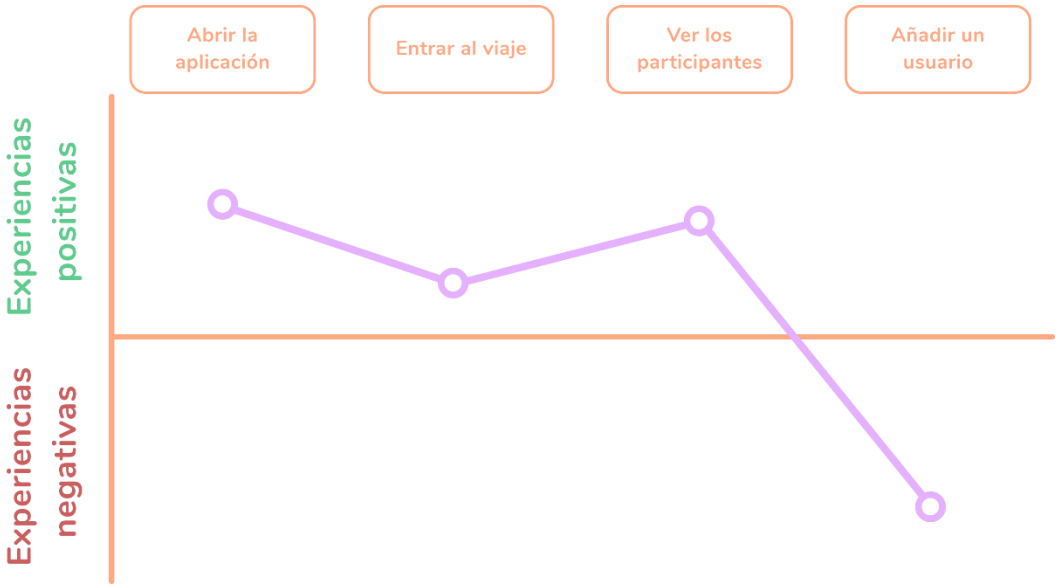


Figura 26. User Journey Map #1.
(Fuente propia)

El segundo *User Journey Map* se basa en eliminar una actividad del itinerario de viaje, añadir una nueva y después editar esta:

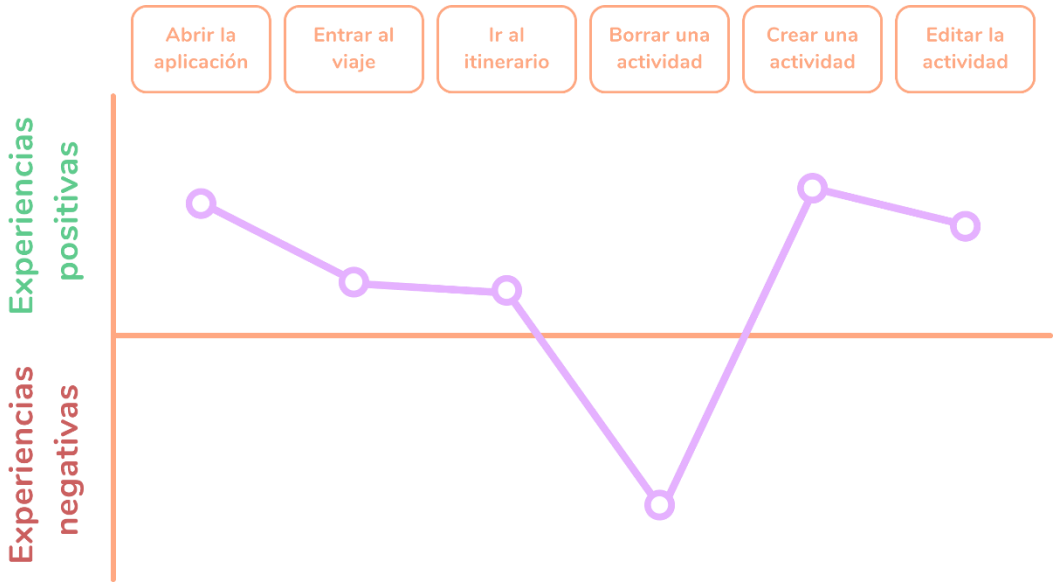


Figura 27. User Journey Map #2.
(Fuente propia)

De ambos gráficos se puede deducir que acciones como abrir la aplicación, ir al viaje concreto, ir a las secciones correspondientes… producen emociones positivas/neutras en los usuarios. Esto se debe a que no se está llevando a cabo ninguna acción concreta que pueda causar desperfectos, daños o pérdidas de información, sino que más bien se está navegando por la aplicación.

Los puntos más negativos son la gestión de usuarios de un viaje y eliminar una actividad. Respecto al primero, un usuario puede sentirse inseguro al seleccionar a quién añadir a un viaje, pues por error podría añadir a otra persona (incluso a alguien que no conoce). Por su parte, el eliminar una actividad del itinerario supone una tensión por si se borra otra sin querer y no se puede deshacer la acción.

Al mismo tiempo, se aprecian experiencias positivas cuando se crea una actividad nueva. El motivo se debe a que el usuario por fin siente cómo se están solucionando sus problemas o satisfaciendo sus necesidades.

Las conclusiones más significativas de estos *User Journey Maps* son que hay que prestar una mayor atención a las acciones destructivas (borrar una actividad, eliminar un ítem del *checklist*…) y a los momentos en los que se debe elegir a usuarios concretos (para añadirlos a un viaje, etc.). De esta forma, se conseguirá que el usuario tenga una mayor confianza y seguridad en sus acciones.

Para mejorar la experiencia de usuario, en esas acciones concretas se deberá mantener una interfaz limpia, cuidada y que aporte *feedback* al usuario, incluyendo una confirmación de la acción si se cree necesario.

Por otro lado, a pesar de no ser tan prioritario como intentar minimizar las experiencias negativas, también sería interesante reforzar los puntos en los que el usuario experimenta emociones positivas. Por ejemplo, se puede felicitar o recompensar al usuario al crear un nuevo viaje o cualquier otro mecanismo que le aporte *feedback*. De esta forma, se destacan las acciones “buenas” frente a las “malas”.

8.5. Guía de estilos

8.5.1. Logotipo

La *app* móvil que solucionará los problemas de organizar viajes en grupo se llamará *veelu*. Este nombre es una combinación de las palabras neerlandesas *veel* (“mucho”) y *u* (“tú”). El significado de *veelu* se puede interpretar como el beneficio que obtienes tú gracias a las muchas herramientas que te ofrece la *app*.

El origen de esta combinación se debe a la experiencia propia de un viaje con amigos a los Países Bajos. En este viaje, volvió a notarse la necesidad de emplear una herramienta para que la planificación fuese mejor. Por ello, se ha decidido que el nombre de la *app* fuera una referencia al viaje que hizo recordar la motivación de este Trabajo de Fin de Grado.

Una vez se ha dejado claro el significado y origen del nombre de la marca, el siguiente paso es el diseño de su logo. Tras innumerables ideas y bocetos, el resultado final es el siguiente:

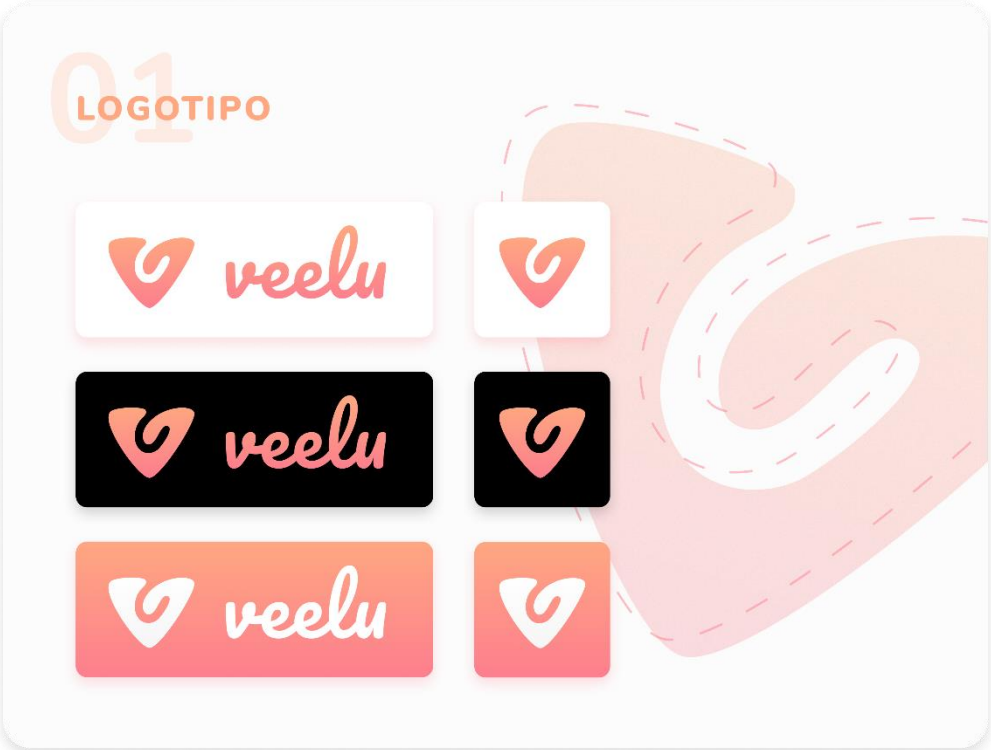


Figura 28. Logotipo.
(Fuente propia)

El logotipo está compuesto por el nombre de la *app* y un icono. Este icono es la combinación de las letras “v” y “u”, pero sin llegar a ser demasiado obvio a primera vista. El resultado es un icono simple, minimalista y que transmite dinamismo.

Este diseño, además, ayudará a la aplicación a desmarcarse de otras similares que ya existen en el mercado. Estas *apps* suelen optar por logotipos basados en elementos relacionados con los viajes, como brújulas, mapas, maletas, aviones, etc. El hecho de que tengan un diseño más tradicional no es algo malo, pero sí es cierto que, al ser tan parecidos entre sí, tienen más difícil destacar y llamar la atención de los usuarios.

8.5.2. Colores corporativos

En cuanto a los colores de la marca, en la siguiente imagen se encuentran los 4 más importantes:

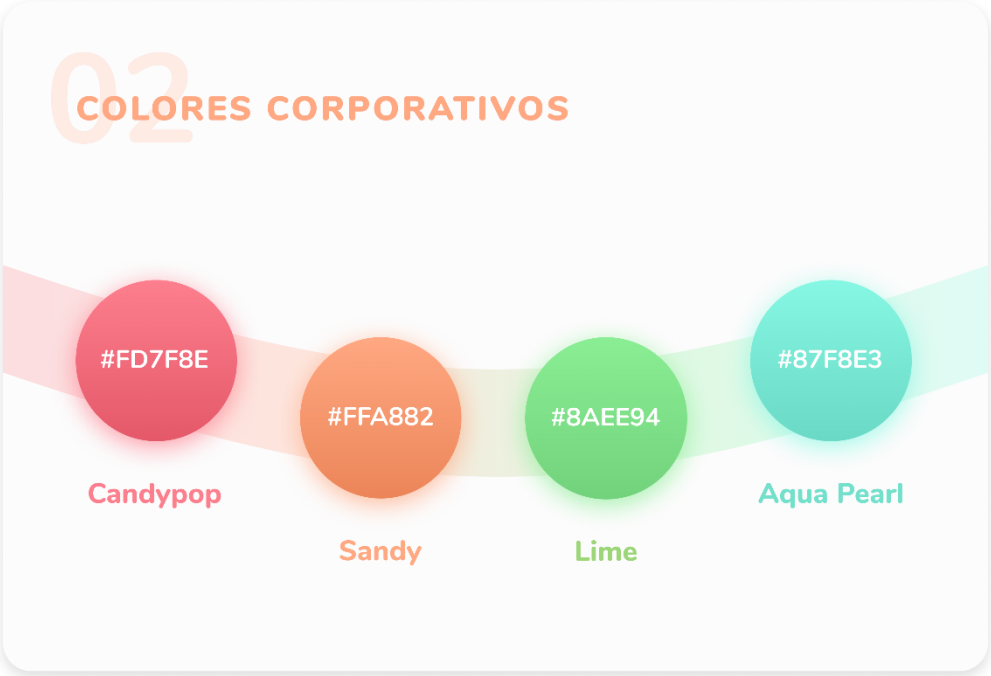


Figura 29. Colores corporativos.
(Fuente propia)

El objetivo que se pretendía era que los colores corporativos aportasen un toque moderno y fresco, por eso se han seleccionado colores vibrantes tanto cálidos como fríos.

Los dos primeros son los que conforman el degradado del logotipo. Todos los colores seleccionados son fácilmente distinguibles entre sí y tienen un buen contraste tanto con textos y fondos blancos como negros [RNF2].

8.5.3. Fuente

El tipo de fuente elegida para *veelu* es la fuente "*Nunito*", de Vernon Adams [29]. Como se aprecia en la imagen a continuación, es una tipografía *Sans Serif*[31], las cuales son más legibles en webs o aplicaciones móvil [30] y se asocian más con marcas juveniles.

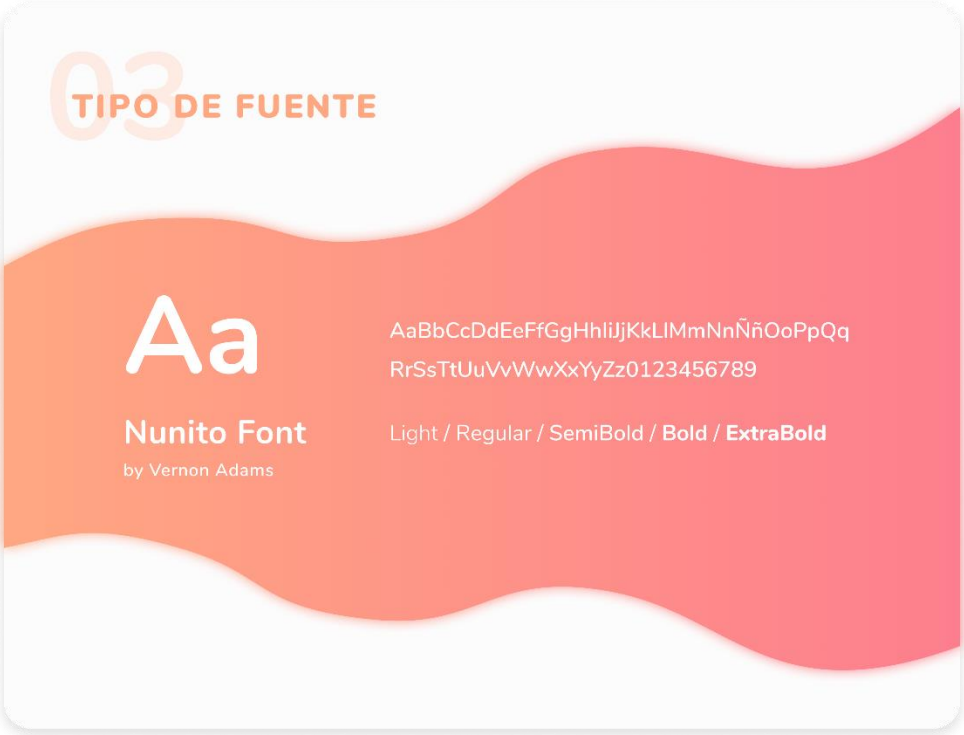


Figura 30. Tipografía.
(Fuente propia)

Otro de los motivos por los que se ha elegido *Nunito* es el alto contraste entre sus caracteres, que hacen que sea muy legible [RNF2]. Además, tal y como se puede comprobar en la Figura 30, *Nunito* ofrece numerosos estilos (*light*, *semibold*...) que se utilizarán para darle mayor o menor importancia a los textos de la aplicación.

[31] Tipografía en la que los caracteres carecen de las terminaciones llamadas remates, gracias o serifas.

8.6. Diseño Interfaces

A partir de esta sección, se comienzan a materializar los requisitos y lo relacionado con la experiencia de usuario. Al diseñar interfaces, se concreta sobre cómo el usuario va a interactuar con la aplicación.

Este apartado es muy importante y merece la pena dedicarle el mayor tiempo posible. Cuantas más interfaces queden diseñadas y bien definidas, más tiempo se ahorrará de cara a la fase de implementación. Esto se debe a que se evita la toma decisiones relacionadas con el diseño, permitiendo centrarse únicamente en programar la aplicación.

Existen varios niveles de fidelidad a la hora de diseñar las interfaces. En primer lugar, están los *wireframes*[32] que, a pesar de ser poco exactos respecto al resultado final, sirven para representar de forma fácil cómo y dónde se ubican los elementos. Se podría considerar la interfaz base que poco a poco se va mejorando.

Si se sube un nivel más de fidelidad, entonces se habla de *mockups*[33]. Este tipo de interfaces sí representan de manera exacta dónde se ubican los elementos y cómo van a verse en el resultado final. En este nivel ya se introducen los colores, formas, tipo de fuente, etc.

Por último, están los prototipos, que son como los *mockups* pero que también simulan la interactividad. Este tipo de diseño es prácticamente la representación del resultado final, pero supone invertir mucho tiempo en la fase del diseño.

En el caso de este Trabajo de Fin de Grado, se ha optado directamente por hacer *mockups*. Como se tiene una visión más o menos clara de lo que quiere diseñar, no se cree necesario diseñar *wireframes*. Tampoco se ha desarrollado un prototipo debido a que supondría invertir demasiado tiempo en este proceso.

Para realizar el diseño de *mockups* se ha utilizado la aplicación *Figma* [31]. En la siguiente imagen, se muestra una vista global de las interfaces que contendrá *veelu*. Si bien es cierto que faltan algunas interfaces, se han diseñado la mayoría de ellas y las más importantes.

[32] Guía visual que representa el esqueleto o estructura visual de un sitio web.
[33] Diseño digital de una web y/o aplicación.

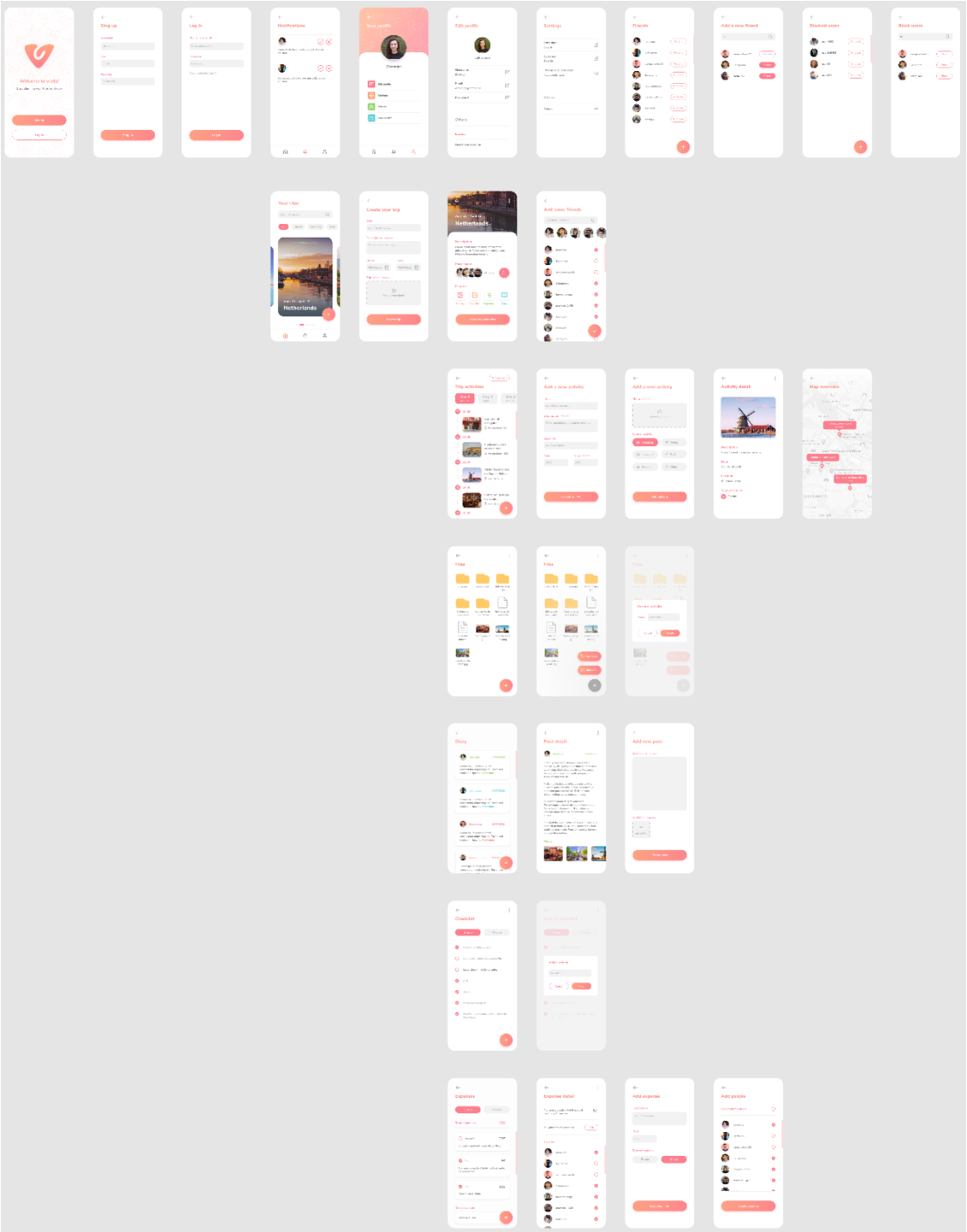


Figura 31. Vista global de las interfaces.
(Fuente propia)

Se han diseñado un total de 32 interfaces para la aplicación, las cuales siguen la guía de estilos definida en el apartado anterior. A continuación, se van a comentar los aspectos más importantes de cada una de ellas. En los subtítulos de cada captura se encuentran los IDs de cada interfaz para hacer referencias a estas más adelante.

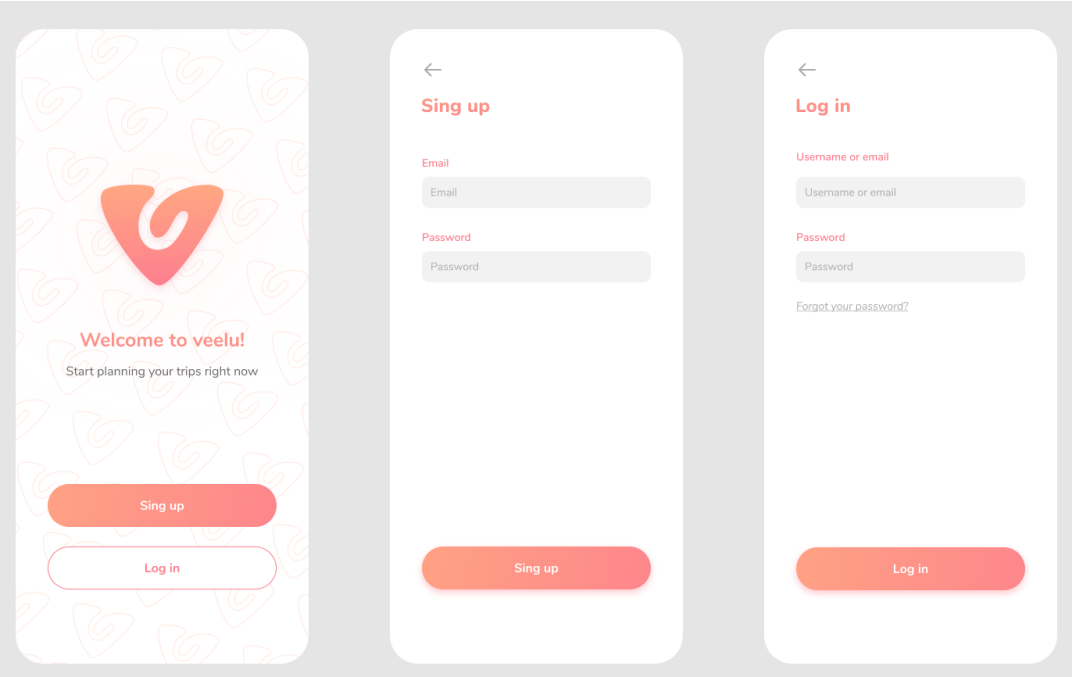


Figura 32. Interfaces 1, 2 y 3.
(Fuente propia)

En primer lugar, se encuentra la primera interfaz con el logo y un mensaje de bienvenida al entrar a la aplicación. A partir de aquí, un usuario podrá iniciar sesión o registrarse. En el registro se ha optado por pedir solo los datos necesarios para agilizar el proceso.

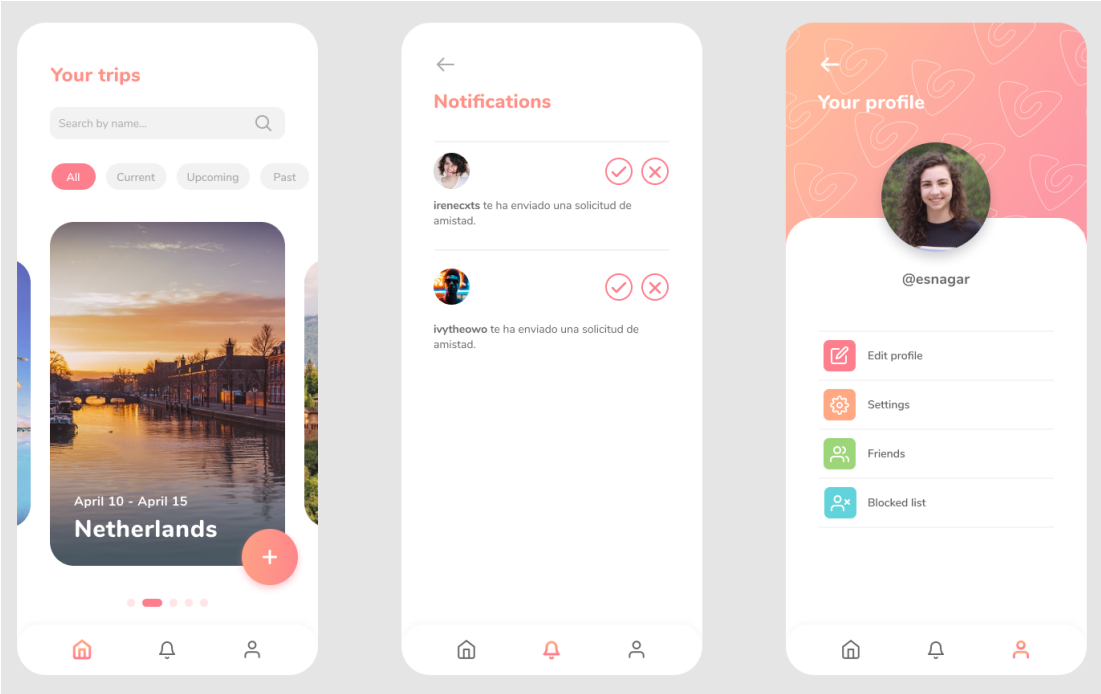


Figura 33. Interfaces 4, 5 y 6.
(Fuente propia)

En la imagen anterior se ven las tres interfaces accesibles desde el menú de navegación. La primera es la interfaz "*Home*", que muestra el listado de viajes de un usuario, pudiendo utilizar los filtros de arriba o el buscador. El botón flotante permite crear un nuevo viaje.

En la siguiente, se muestran las notificaciones que tiene un usuario. En concreto, estas informan sobre peticiones de amistad. Por último, la tercera interfaz muestra el perfil de un usuario. Desde aquí se podrá acceder a funcionalidades como los ajustes, gestión de amigos, etc.

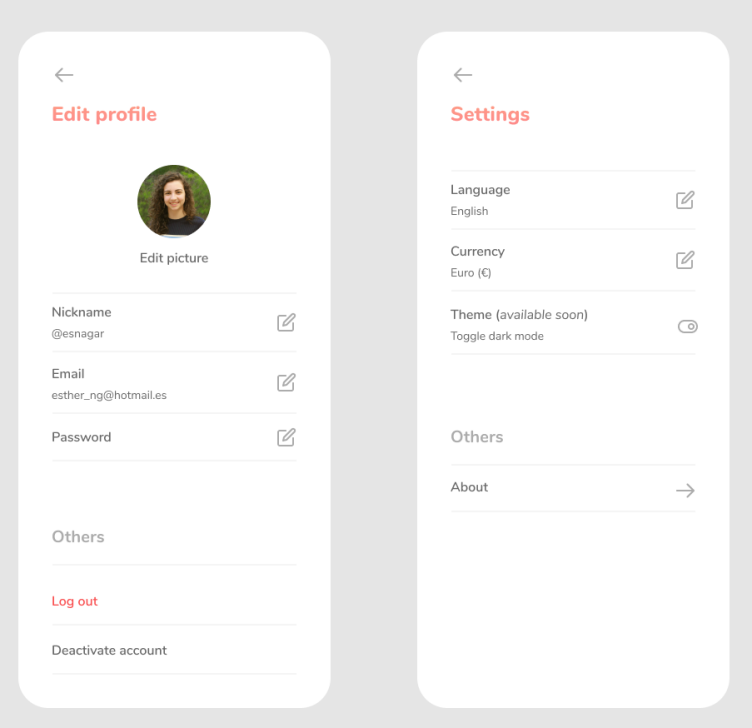


Figura 34. Interfaces 7 y 8.
(Fuente propia)

Ahora se continua con las interfaces relacionadas con los datos de un usuario. En esta captura, aparecen las interfaces correspondientes a los dos primeros enlaces de la interfaz [ID6]. En "*Edit profile*" se pueden modificar datos básicos, cerrar sesión e incluso eliminar la cuenta.

Desde "*Settings*", por otro lado, se pueden editar aspectos relacionados con la *app* en sí, como el idioma, el tipo de moneda o la apariencia. También hay un apartado de "*About*", que mostrará un pequeño texto explicando quién ha hecho la *app* y por qué.

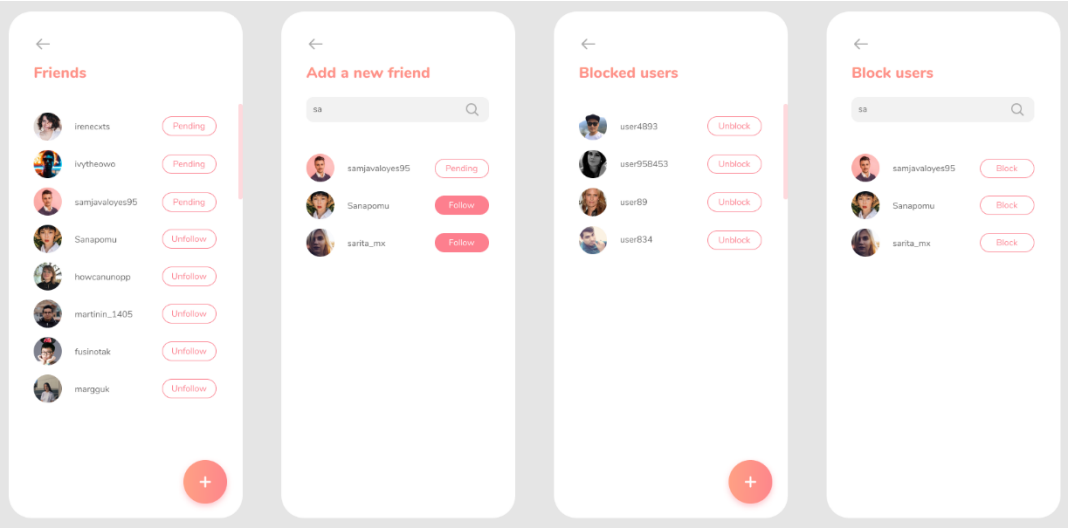


Figura 35. Interfaces 9, 10, 11 y 12.
(Fuente propia)

Desde el perfil de un usuario también se accede a la interfaz que lista los usuarios amigos (y a los que no han aceptado la petición aún) y a la interfaz que lista los bloqueados. Al pulsar en el botón flotante de la interfaz de bloqueados, redirige a otra que permite buscar usuarios por su *nick* y bloquearlos. El proceso es el mismo para añadir usuarios amigos.

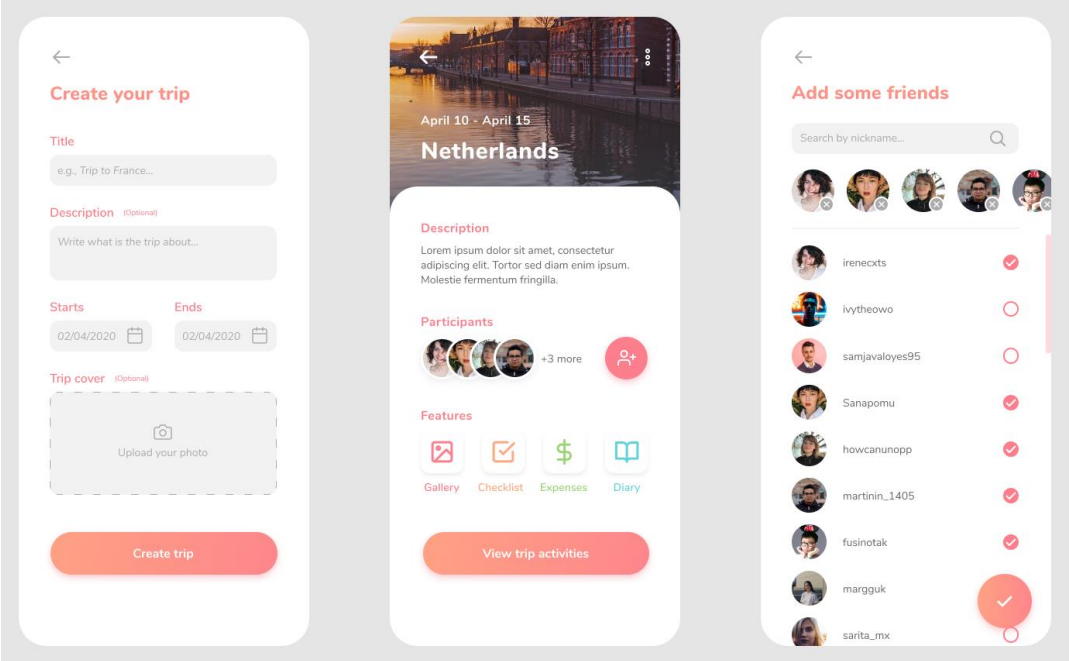


Figura 36. Interfaces 13, 14 y 15.
(Fuente propia)

A continuación, se va a hablar de las interfaces relacionadas con los viajes. La primera de ellas consiste en el formulario para rellenar la información necesaria de un nuevo viaje.

La segunda interfaz es la que tiene todo el detalle de un viaje y, además, contiene muchas de las funcionalidades recogidas en el capítulo de los requisitos. Los administradores podrán gestionar qué usuarios participan en él con el botón que aparece a la derecha.

En el capítulo de análisis de la experiencia de usuario, se vio que gestionar los participantes puede generar emociones negativas. Para evitar esto, se ha diseñado una interfaz únicamente para esta tarea. Además, solo se puede añadir a un usuario si se es amigo de este (en el peor de los casos se añadiría a un amigo por error, menos grave que añadir a un desconocido).

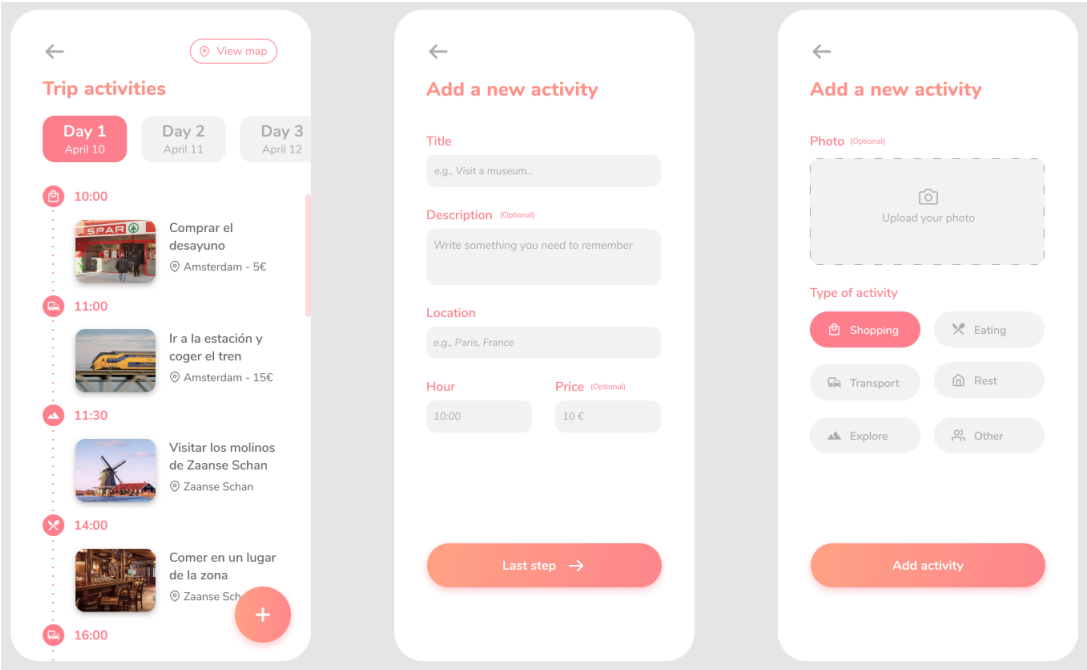


Figura 37. Interfaces 16, 17 y 18.
(Fuente propia)

En la segunda interfaz de la Figura 36, al darle a "*View trip activities*" se accede al listado con las actividades planeadas para el viaje, organizadas por día. Cada una de estas actividades tiene una hora de inicio, título, precio, tipo de plan, etc.

Si se desea añadir una nueva, será necesario pulsar el botón flotante y rellenar el formulario de la derecha. La acción de crear una actividad se ha dividido en dos interfaces, ya que en una sola habría quedado muy recargado.

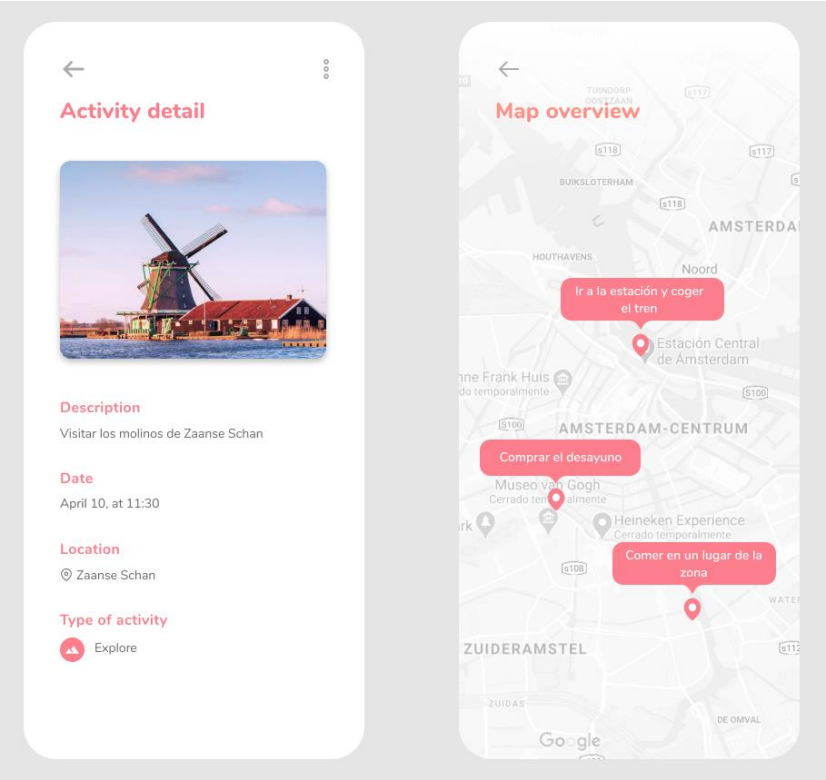


Figura 38. Interfaces 19 y 20.
(Fuente propia)

En la imagen interior se observa, en primer lugar, la interfaz del detalle de una actividad con toda su información. Al igual que el resto de interfaces que muestran un elemento en detalle (viaje, actividad, entrada de diario, etc.), el icono de la parte superior derecha mostrará un desplegable que permitirá editar o eliminar dicho elemento.

Por otro lado, en la segunda interfaz se visualiza una vista general de la ubicación de las actividades en un mapa. En la interfaz se ha utilizado una captura de *Google Maps* como ejemplo, pero en su lugar se usará *Mapbox* para proveer un mapa más minimalista. A esta interfaz se accede desde el botón superior derecho en la interfaz del listado de actividades.

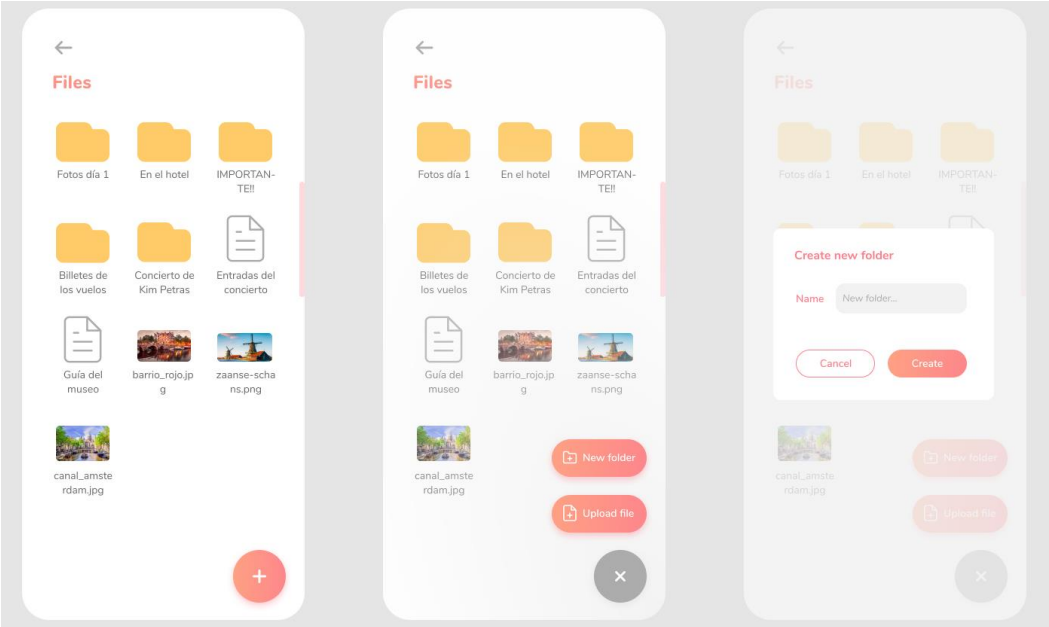


Figura 39. Interfaces 21, 22 y 23.
(Fuente propia)

A partir de aquí, se muestran las interfaces relacionadas con las funcionalidades de un viaje (guardar archivos, diario, *checklist* y gastos, en este orden). Estas tres interfaces muestran el listado de carpetas y archivos de un viaje. Si se pulsa el botón flotante, se podrá crear una carpeta o subir un archivo desde el dispositivo móvil.

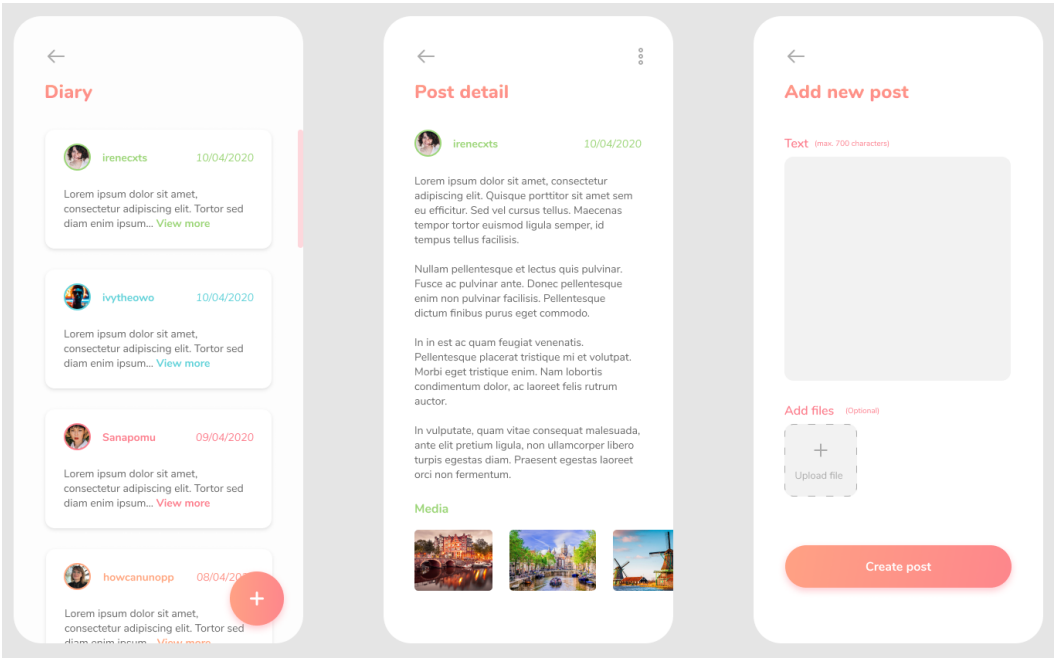


Figura 40. Interfaces 24, 25 y 26.
(Fuente propia)

Estas pantallas sirven para escribir entradas de diario. La primera lista las publicaciones de los participantes, la segunda muestra el detalle de una y la tercera sirve para subir una entrada.

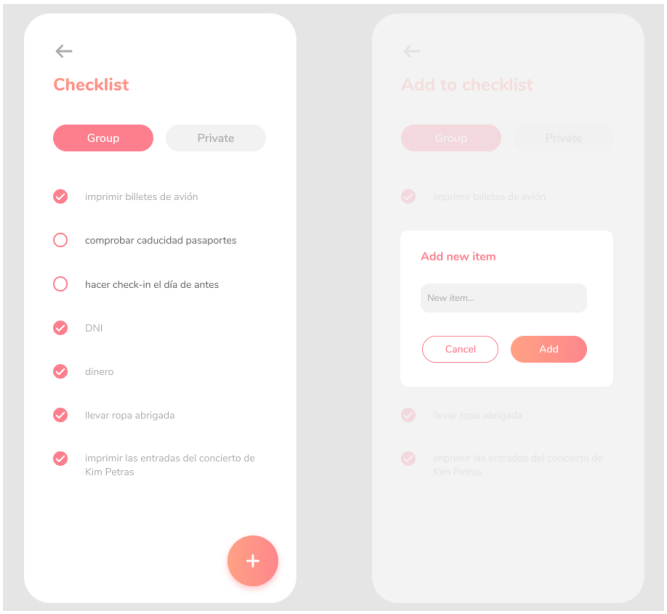


Figura 41. Interfaces 27 y 28.
(Fuente propia)

Estas dos interfaces son las destinadas a mostrar el *checklist* y añadir ítems. Este *checklist* se puede aprovechar para apuntar qué objetos llevar al viaje, qué cosas hay que hacer antes de viajar... Se pueden anotar ítems tanto en la zona grupal como de forma individual.

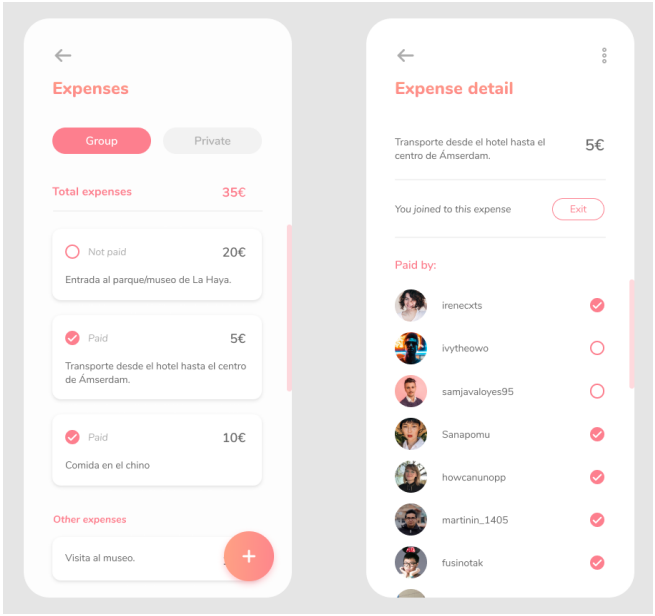


Figura 42. Interfaces 29 y 30.
(Fuente propia)

Ahora se enseñan las interfaces de los gastos de un viaje. A la izquierda se encuentra la que lista cada uno de los gastos (pudiendo indicar si se han pagado ya o no) y la suma del total de las expensas. En esta sección también se distingue entre lo privado y lo grupal, pudiendo anotar gastos individuales. Debajo de "*Other expenses*", aparecen gastos de los que el usuario autenticado no forma parte (así siempre podrá unirse a ellos).

Si se accede al detalle de uno de los gastos, se podrá consultar información básica y el listado de usuarios asociados a ese gasto, mostrando si lo han pagado o no. Solo la propia persona puede indicar si él o ella ha pagado ese gasto. Si se quiere dejar de tener relación con ese gasto, tan solo se deberá pulsar el botón "*Exit*".

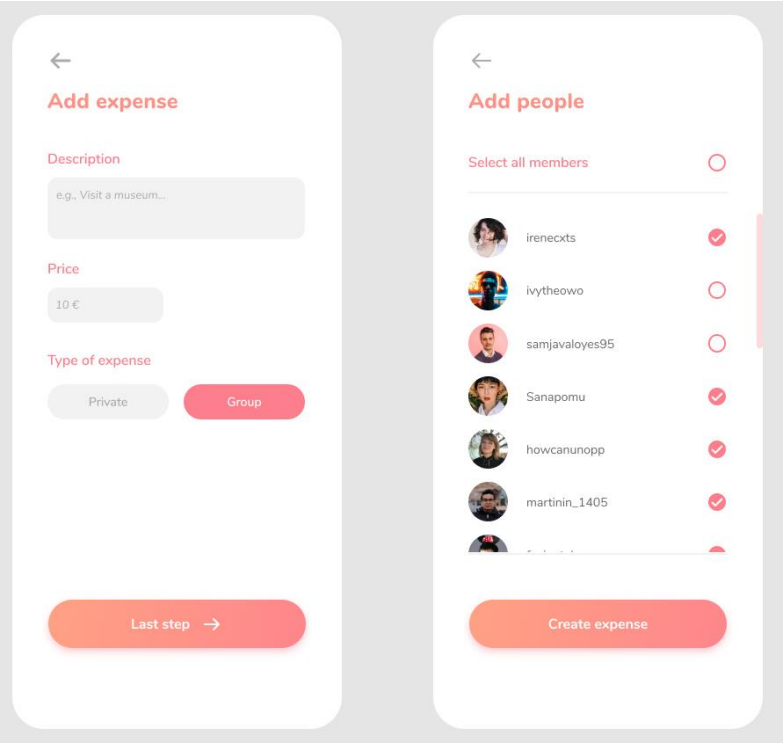


Figura 43. Interfaces 31 y 32.
(Fuente propia)

Estas son las últimas interfaces. Mediante el formulario de la primera de ellas se podrá añadir un gasto. Se ha de especificar una descripción, precio y si el gasto es privado o no. En caso de ser grupal, saldrá un listado que permite asociar usuarios a ese gasto.

Como se puede observar, se le ha dado mucha importancia a la consistencia entre interfaces. Al repetir los mismos elementos (el botón para crear, el menú de editar/eliminar dentro del

detalle...), un usuario puede familiarizarse y asociar conceptos más rápido [RNF1]. También se ha hecho hincapié en el diseño minimalista, directo y sin sobrecargar al usuario.

Estas 32 interfaces analizadas son la base de la aplicación, pero la *app* final contendrá muchas más interfaces que no se han contemplado en esta sección. Por este motivo y por la limitación de tiempo del TFG, es recomendable definir un orden de prioridad de interfaces, en el caso de que no dé tiempo a implementarlas todas.

Para este proyecto, las interfaces más prioritarias son las de la autenticación, los viajes y el perfil de usuario. En un segundo nivel están aquellas relacionadas con las funcionalidades especiales de los viajes (archivos, diario, *checklist*, gastos...). Por último, si se dispone del tiempo suficiente, se implementarán la de bloquear usuarios, visualizar ubicaciones en el mapa, etc.

8.7. Diseño de pruebas y validación

Como último subapartado en este capítulo se encuentra el diseño de pruebas y validación. Estas pruebas permiten comprobar el correcto funcionamiento de la *app* y también indican qué partes necesitan ser arregladas o mejoradas.

En el caso de un Trabajo de Fin de Grado, lo más importante es comprobar si se han alcanzado los objetivos marcados al principio. Para ello, se van a detallar las diferentes pruebas y métodos que se emplearán para validar la aplicación.

El primero de ellos, el más simple, consiste en comprobar si se han desarrollado y se cumplen cada uno de los requisitos detallados. Aunque pueda parecer una prueba innecesaria, es posible que algún requisito quede sin cumplirse del todo o que se haya olvidado, así que nunca está de más hacer esta comprobación.

Por otro lado, se hará uso de los productos de pruebas de calidad de Firebase [32], como *Crashlytics* (para generar informes de fallos), *Performance* (para las métricas de rendimiento y latencias que experimentan los usuarios) o *Test Lab* (para hacer pruebas).

El servicio de *Test Lab* se utiliza desde la propia plataforma de Firebase. Por otro lado, para beneficiarse de los otros productos, se han de añadir sus SDK al proyecto. En el caso de *veelu*, solo se realizarán estas pruebas para *Android*.

Además de utilizar estos productos de *Firebase*, se podría hacer *testing* propio. Por desgracia, *Ionic* no aporta ninguna herramienta que facilite este proceso, por lo que supondría invertir demasiado tiempo en este apartado, el cual no es tan importante tratándose de un TFG.

Debido a estos motivos, se ha decidido invertir el tiempo que se dedicaría a hacer pruebas unitarias a pulir la aplicación a partir de las opiniones de personas ajenas al proyecto.

Para conocer estas opiniones, es recomendable apoyarse en la *Play Store*. La gente será capaz de descargar la *app* a través de esta plataforma, probarla y dejar su opinión (puntuando del 1 al 5 y/o a través de los comentarios). No obstante, este procedimiento para recibir *feedback* puede resultar insuficiente si no se utiliza junto a otros métodos.

Por este motivo, además de recoger *feedback* de la *Play Store*, se harán unas pruebas controladas para conocer las opiniones de distintas personas como familiares, amigos, etc. Una vez se haya desarrollado la *app* y se disponga del primer prototipo, un grupo de personas de confianza probarán la aplicación y se recogerán sus opiniones.

Además, se intentará que haya una gran variedad dentro las personas elegidas para probar la aplicación: jóvenes, gente mayor, personas con problemas de vista... De esta manera, se podrán conocer puntos de vista muy distintos, los cuales ayudarán a mejorar notablemente la aplicación final.

Para poder obtener la mayor cantidad de información y de la forma más detallada posible, se utilizará *Google Forms* [33]. Esta aplicación permite generar encuestas online, así que se redactará un formulario con todo tipo de preguntas (de puntuar con un número, de elegir entre varias opciones, de redactar con tus propias palabras...). Otro objetivo es que el formulario no sea demasiado largo, sino que debe ser lo más ameno posible.

A continuación, se recogen en una tabla todas las preguntas que aparecerán en dicho formulario, con sus características particulares:

Tabla 7. Formulario para obtener feedback.

Nº	Pregunta	Respuesta
1	Nombre (opcional)	Escribir
2	Edad	Escribir
3	¿Qué smartphone utilizas? (marca y modelo, ej.: *Xiaomi Mi 9*)	Escribir
4	Si has utilizado la *app* en el exterior (donde haya mucha luz), ¿has tenido dificultades para visualizar su contenido?	Sí / No / No la he utilizado en exteriores
5	¿Los colores de la aplicación te han supuesto un problema? (te cuesta distinguir los elementos, no hay suficiente contraste, etc.)	Sí / No
6	Indica si te ha quedado claro cómo funcionan los siguientes apartados: inicio de sesión y registro, perfil de usuario, gestión de viajes, actividades de un viaje, archivos, diario, *checklist*, gastos.	Puntuar del 1 (muy confuso) al 5 (muy claro)
7	Puntúa del 1 al 5 lo fácil que te ha resultado realizar las siguientes acciones: iniciar sesión, registro, editar datos de tu perfil, añadir amigos, crear viajes, elegir una foto desde tu galería, editar una entrada de diario, quitar a usuarios dentro de un viaje, etc.	Puntuar del 1 (muy difícil) al 5 (muy fácil)
8	¿Qué cosas te han gustado de la aplicación? (opcional)	Escribir
9	¿Qué es lo que menos te ha gustado de la aplicación? (opcional)	Escribir
10	Indica cuáles son los aspectos de la *app* que se deberían arreglar/mejorar: accesibilidad (más contraste, mayor tamaño de los elementos…), usabilidad (que se entienda más rápido cómo funciona, que sea más intuitiva…), fiabilidad (que no de errores), estética, rendimiento (que vaya más rápido), otro (escribir).	*Checklist*

Como se puede comprobar, es un formulario corto y que abarca todo tipo de cuestiones sobre la aplicación (funcionamiento, errores, qué se debería mejorar o arreglar…).

Gracias a este método de pruebas, se podrá conocer la opinión de usuarios reales sobre aspectos concretos de la aplicación. Esto permitirá invertir más tiempo en arreglarla y pulirla, para que el resultado final sea el mejor posible.

9. Implementación

Tras definir los problemas y diseñar la solución, es el momento de comenzar con la implementación del proyecto. Este apartado recoge todo el proceso y tareas que se han llevado a cabo para desarrollar la aplicación móvil *veelu*. Además, se comentan también los contratiempos que han surgido durante la implementación y cómo se les ha dado solución.

En el apartado de metodología, se escogió una adaptación de *Scrum* para organizar tareas y definir tiempos. Por ello, se muestran los avances por *sprints*, cuya duración es de una semana.

Como dato a tener en cuenta, pese a haber finalizado el curso académico, el desarrollo de la *app* coincidió con la realización de la asignatura prácticas en empresa, lo cual ha supuesto una restricción de tiempo para trabajar en el TFG. Por este motivo, cada *sprint* se traduce en unas 25 horas semanales, trabajando por las tardes y los fines de semana.

9.1. Sprint 1: preparación y proyecto base

La primera semana no se implementó nada, sino que tuvo el objetivo de preparar el entorno de desarrollo y todo lo necesario para comenzar con este proceso.

En cuanto a las herramientas y programas utilizados, se ha empleado *Visual Studio Code* [34], editor de código gratuito desarrollado por *Microsoft*. Para el control de versiones, se ha utilizado *GitHub* junto a *GitKraken* [35], que facilita esta gestión gracias a sus herramientas e interfaz gráfica. Para comprobar el contenido de la base de datos, se ha utilizado la propia web de *Firebase*.

Por otro lado, se han ido visualizando los cambios realizados en la *app* con el navegador *Brave* [36] (ya que *Ionic* abre un servidor *localhost*). Por otro lado, de vez en cuando se iba comprobando que funcionase de la misma forma en un dispositivo móvil. Para ello, *Ionic* necesita que esté instalado el IDE *Android Studio* [37] y que haya un *smartphone* conectado al ordenador vía USB. En concreto, las pruebas se han llevado a cabo con un *Xiaomi Redmi Note 8*, un *Xiaomi Mi A3* y un *Huawei P30 PRO*.

Volviendo con el *sprint* en sí, los primeros días se dedicaron a repasar los conceptos básicos de las tecnologías a utilizar y a hacer pequeñas pruebas para afianzar dichos conocimientos.

Después de haber leído varios artículos y seguir algunos tutoriales, se comenzó con la creación del proyecto base. Para ello, se instaló *Ionic* y se creó un proyecto basado en *Angular*, al cual se le añadió *Capacitor* [38], como recomienda la propia documentación de *Ionic*. *Capacitor* es, en pocas palabras, un nuevo *Apache Cordova* pero desarrollado por el propio equipo de *Ionic*.

El siguiente paso fue añadir un menú de *tabs*[34], crear tres páginas de ejemplo y configurar el *routing*[35] de la aplicación para navegar entre ellas. Una vez creada la base, se continuó con la instalación de librerías. Se añadió el paquete de *Tailwind CSS* [39], un *framework* de *CSS* de bajo nivel que permite customizar elementos de forma muy rápida.

Por otro lado, se instaló *AngularFire* [40], la librería oficial de *Firebase* para trabajar fácilmente dentro de *Angular*. Lo más destacable de *AngularFire* es que está basada en *RxJS* [41], una de las librerías de *JavaScript* más populares hoy en día.

RxJS permite manejar eventos y colecciones de forma asíncrona. Su elemento básico es el llamado *Observable*. Un *Observable* es una fuente de datos que pueden llegar con el tiempo (normalmente se habla de eventos). Luego están los *Subscribers*, que se asocian a estos *Observables*. Cuando un *Observable* detecta el evento, llama a su *Subscriber*, que es el encargado de ejecutar el código que se le indique.

No obstante, también están los *Operators*, que manipulan los datos recibidos antes de llegar al *Subscriber* y devuelven un *Observable* de estos valores transformados. Por ejemplo, permiten filtrar la información o guardarla en una estructura de un mapa.

Todos estos conceptos pueden resultar algo complejos al principio (y se complica mucho más), pero es necesario comprender la base de *RxJS* para trabajar con *AngularFire*. En definitiva, el funcionamiento es el siguiente: se tiene un *Observable* asociado a un evento y, cuando este ocurre, se captura el objeto *Observable*, se transforma con los *Operators* y, por último, se trata su contenido a través de su *Subscriber*.

[34] Menú con varias pestañas.
[35] Enrutamiento, cómo se navega por las páginas.

Mediante *RxJS*, la librería *AngularFire* trata las peticiones a *Firebase* de forma asíncrona y con más facilidad que si se hiciera con *JavaScript* puro. Y lo más importante es que, a través de los *Observables*, *AngularFire* ofrece actualizaciones de cualquier cambio en tiempo real.

Esta última característica es vital para la aplicación a desarrollar, puesto que el objetivo es que pueda ser utilizada por varias personas al mismo tiempo. Gracias a *AngularFire*, cualquier cambio realizado por un usuario dentro de un viaje se reflejará de forma instantánea para el resto de usuarios asociados a este.

En definitiva, en este primer *sprint* se hicieron pequeñas pruebas, se investigó las bases de las tecnologías a usar y se preparó el proyecto comenzar con la implementación.

9.2. Sprint 2: customización y primeras interfaces

Antes de empezar con el desarrollo, se crearon variables en la hoja de estilo global para definir los colores de marca. Como *Ionic* utiliza un preprocesador de *CSS* llamado *Sass*, se pueden crear variables con los colores corporativos y utilizarlos en cualquier lugar de la hoja de estilos.

De esta forma, si en un futuro se decide cambiar o modificar alguno de ellos, no supondría trabajo alguno. A partir de esos colores, se hicieron varias clases para definir el color de fondo de los elementos, su borde, el color del texto, etc.

Por otro lado, se descargaron de *Google Fonts* los archivos TTF de *Nunito* (el tipo de fuente a utilizar en la *app*), se añadieron en una carpeta y se crearon clases para definir los distintos tipos de letra (normal, negrita, fina...).

Además, se descargaron algunos de los iconos que iban a formar parte de la aplicación. En concreto, se utilizaron los de la colección de *Feather Icons* [42], *open source* y con un estilo minimalista. En lugar de descargar todos los iconos, se fueron añadiendo al proyecto los archivos SVG según se iban necesitando.

Ahora sí, ya se pasa a hablar del desarrollo. Un factor que se ha tenido muy en cuenta es la modularidad del código, es decir, crear elementos y componentes que se reutilicen en distintas

partes de la aplicación. De esta forma, no se tienen que retocar una a una todas las páginas donde se utilice dicho elemento, sino que solo hace falta modificar el propio componente.

El siguiente paso fue, entonces, analizar y definir los elementos que se repiten a lo largo del diseño. Por ejemplo, se puede observar la flecha de volver a la pantalla anterior, el título de la página, las etiquetas de los formularios, la barra de búsqueda, los botones de los filtros, etc. Todos estos elementos se definieron en pequeños componentes y se fueron añadiendo a los módulos de las páginas que los utilizan.

En la siguiente captura, se muestra un ejemplo de cuáles son los elementos que tienen en común 3 interfaces distintas. Se ha destacado con el mismo color la mayoría de los elementos que se repiten (no se han señalado todos para no sobrecargar la imagen):

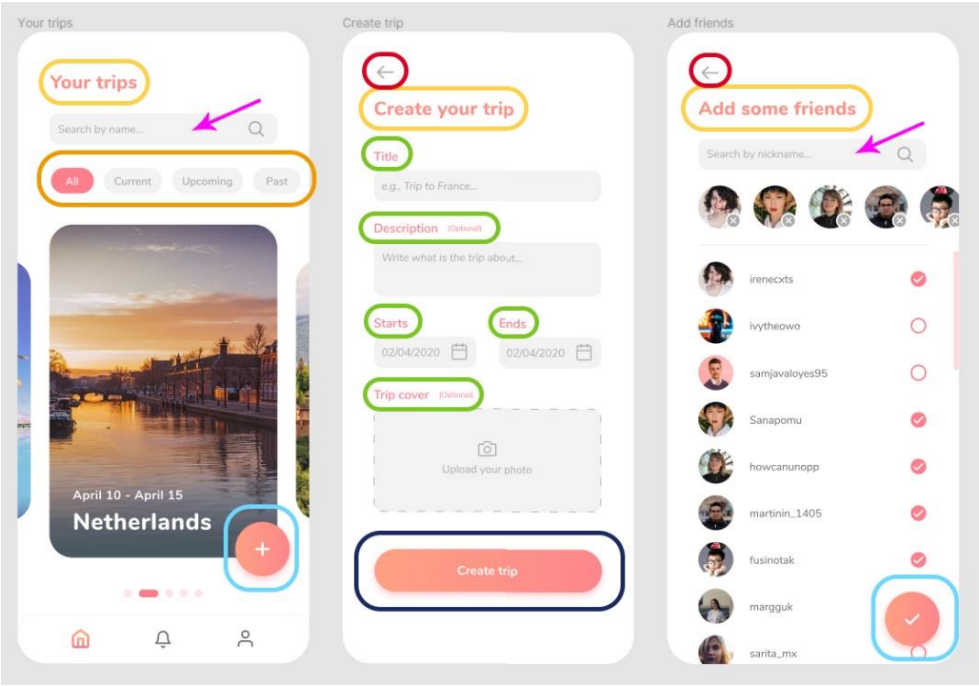


Figura 44. Análisis de elementos reutilizables.
(Fuente propia)

Para almacenar los componentes, páginas, la lógica de la aplicación... se crearon carpetas para tener organizado el código. La estructura del proyecto es la típica distribución que se encuentra en la mayoría de proyectos de *Ionic* con *Angular*:

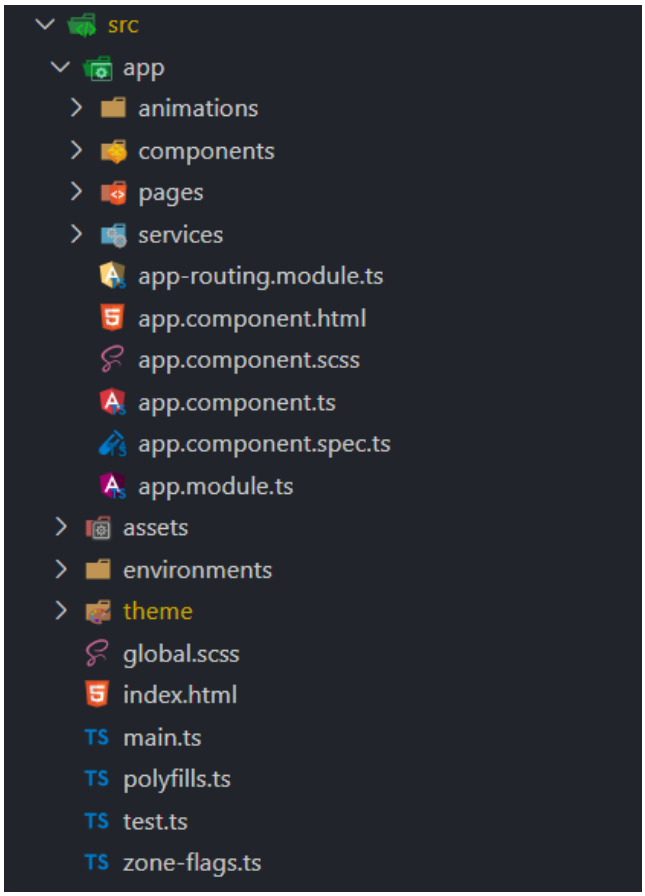


Figura 45. Estructura de archivos del proyecto.
(Fuente propia)

El primer objetivo en este *sprint* era implementar la pantalla del listado de viajes de un usuario [ID4]. Para ello, se generaron los componentes que la forman, como el título de la página, el buscador, los botones de filtros, el botón flotante y la miniatura de la imagen.

Cabe destacar de que, a pesar de que *Ionic* ofrece una gran variedad componentes predefinidos, la mayoría de veces se ha optado por diseñarlos desde cero. Esto se debe a lo difícil que resulta sobrescribir y modificar su estilo ya definido, que deriva en perder mucho tiempo si de desea personalizar a fondo dichos componentes.

No obstante, sí se han utilizado algunos de estos elementos predefinidos en ciertas ocasiones. Es el caso de la pantalla de listados de viajes. Tras crear unos objetos "viaje" de ejemplo, se necesitaba poder ir visualizándolos de uno a otro, al deslizar el dedo por la pantalla. Para ello, se utilizó el componente llamado *ion-slider*, que hace que este deslizamiento tenga una animación muy natural.

Una vez se perfeccionó del todo la interfaz del listado de viajes, se comenzó con la maquetación del formulario para crear uno [ID13]. Al pinchar en el botón flotante, redirige a esta página. Se creó un componente para las etiquetas del formulario y el botón principal que envía su contenido. En el caso de los campos de fecha, se usaron los predefinidos de *Ionic* (*ion-datetime*).

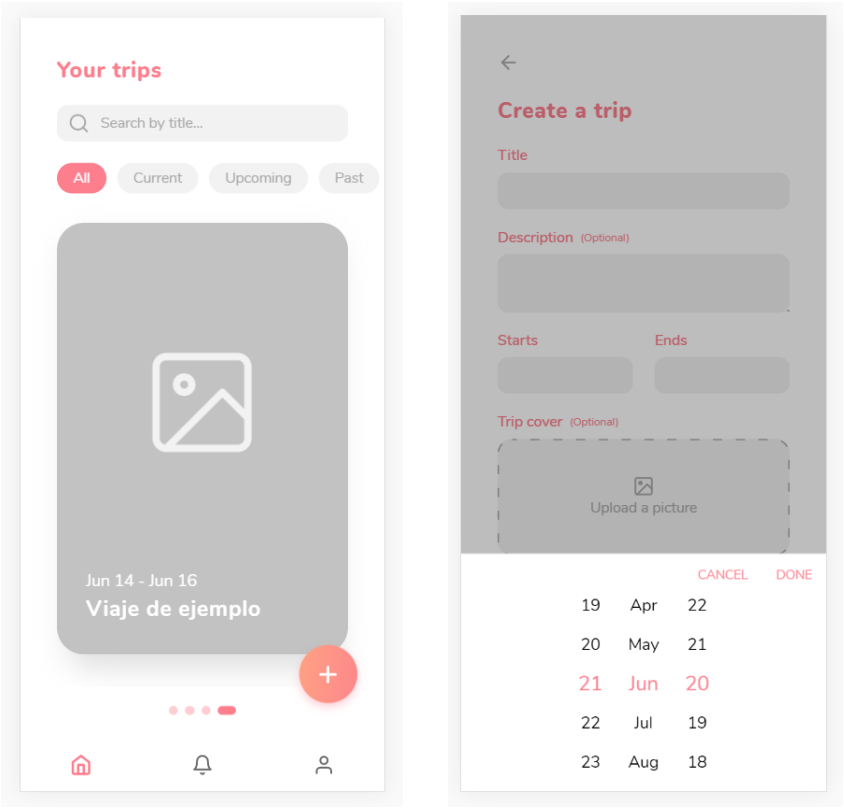


Figura 46. Interfaces de listado de viajes y creación de un viaje.
(Fuente propia)

Además, se continuó con la validación del formulario antes de enviarlo: que estuviesen rellenados los campos requeridos y que las fechas introducidas fuesen válidas. Si no se cumplen estos requisitos, se muestra un mensaje informando del error.

Estos fueron los avances del *Sprint* 2. En resumen, una semana para generar los componentes básicos, páginas, modificar el *routing* de la *app*... y descubrir cómo se trabaja con *Angular* durante el proceso.

9.3. Sprint 3: conexión con Firebase y pruebas en móvil

En este *sprint* por fin se comenzó con la integración de *Firebase* en la *app*. En primer lugar, se creó el proyecto en *Firebase* y se cogieron las credenciales que la aplicación necesitaba. Estas se añadieron en una variable de entorno para, después, inicializar *Firebase* en el módulo raíz de la aplicación. También se añadieron a este módulo los paquetes necesarios para utilizar la *Cloud Firestore* (base de datos *NoSQL*), *Storage* (almacenamiento de archivos) y *Authentication* (autenticación y registro). De estos dos últimos se habla más adelante.

Una vez preparada la conexión con *Firebase*, el siguiente paso fue crear un *service*. Los *service* son un tipo de archivo de *Angular* encargados de incluir e inyectar la lógica de la aplicación en las páginas que lo necesiten. En este caso, se creó un *service* para la gestión de los objetos tipo "viaje": crear, listar, buscar, editar y eliminar.

Para ello, se siguió la propia guía de *AngularFire.* En principio, como todavía no se tenían usuarios autenticados, simplemente se listaban todos los viajes existentes en la base de datos. Para conseguir esta información, se tuvo que indicar el nombre de la colección en cuestión (en este caso, "viajes") y transformar el *Observable* recibido a una estructura de mapa.

Para modificar la información que se muestra en el *HTML* en *Angular* se utilizan los *pipe*. Existen *pipes* para modificar fechas, mostrar todo el texto en minúsculas… aunque el que interesaba en esta ocasión es el de *async*. Este *pipe* muestra el valor de una promesa[36]. En el caso de la *app*, esto quiere decir que muestra el contenido del *Observable* (los viajes en sí) en pantalla.

Una vez listados los viajes, el siguiente paso era pinchar en uno de ellos y verlo en detalle. Cuando se hizo el diseño de la base de datos, se planteó la posibilidad de hacer dos colecciones de los viajes: una más reducida, solo con la información que se muestra en las miniaturas del listado, y otra completa, con todos los datos del viaje. De esta forma, se descargaría menos información.

Sin embargo, los únicos campos que se ahorraría incluir serían el de descripción y el mapa con los usuarios participantes. Así que se decidió que no merecía la pena crear esta versión reducida, la cual supondría más trabajo si se modifica algún dato de un viaje.

[36] Objeto que representa la terminación o el fracaso de una operación asíncrona.

Esta decisión de diseño provocó que, al listar los viajes, se obtuviese toda la información de cada uno de ellos. La ventaja es que, al pinchar para ver el detalle de uno concreto, ahorra tener que hacer otra petición a *Firebase*. En su lugar, lo que se hace es pasarle mediante el *routing* dicho objeto "viaje", con todos sus datos ya cargados.

A continuación, se trabajó en la maquetación del detalle de un viaje [ID14]. Este proceso fue relativamente rápido gracias, de nuevo, a trabajar con componentes reutilizables.

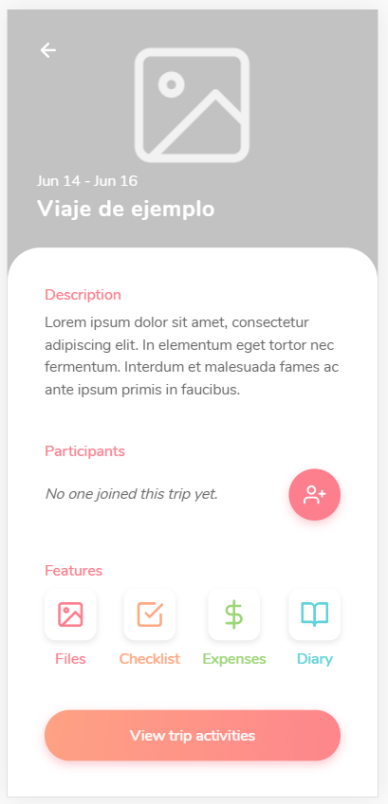


Figura 47. Interfaz de detalle de un viaje.
(Fuente propia)

Después de acabar la interfaz, se desarrolló la funcionalidad de crear un viaje y almacenarlo en la *Cloud Firestore*. La maquetación ya estaba hecha, así que se trabajó en recoger el contenido del formulario y estructurarlo para que coincidiese con los campos de la base de datos. Además, en caso de no escoger una imagen como portada del viaje, añade una por defecto.

Por último, en este *sprint*, se probó la aplicación en un móvil (hasta entonces las pruebas y los cambios se veían en el navegador *Brave*). Para ello, se instaló *Android Studio* como requiere *Capacitor*. Conectando un móvil por USB y generando la *app* con los comandos propios de *Capacitor*, se pudo visualizar la aplicación en un dispositivo *Android*.

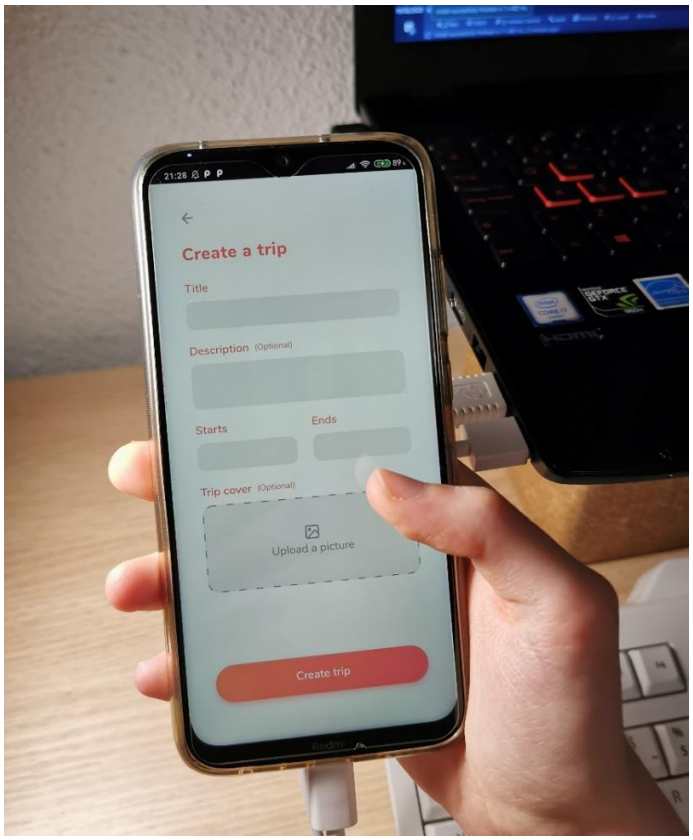


Figura 48. Probando la app en un Xiaomi Redmi Note 8.
(Fuente propia)

Por lo general, todo tenía el mismo aspecto que en la web. Sin embargo, los elementos de la aplicación se desplazaban hacia arriba al pinchar en algún campo y aparecer el teclado del móvil. Tras investigar, se descubrió que esto se debe a la propiedad *windowSoftInputMode*, que se encuentra en el *Android Manifest*[37]. Por defecto, esta tiene el valor de *stateUnspecified*, que se traduce en que es el sistema quien elige cómo se posicionan los elementos, si se mueven o se quedan en su posición original.

Para evitar que los elementos de debajo de la pantalla fuesen empujados por el teclado, se cambió este valor a *adjustPan*, que evita la redimensión y hace que el teclado esté por encima de dichos elementos. Tras esta modificación, la *app* funcionaba como se esperaba.

[37] Archivo que describe información esencial de la aplicación para las herramientas de creación de *Android*, el sistema operativo *Android* y *Google Play*.

9.4. Sprint 4: archivos en Storage y autenticación de usuarios

A pesar de que la creación de viajes funcionaba perfectamente, había un detalle que todavía no se había implementado: elegir una foto como portada del viaje desde la galería del móvil. Por lo tanto, era el momento de investigar sobre *Firebase Storage*, donde se almacenan los archivos como fotos, PDF, vídeos, etc.

No obstante, antes de llegar a ese punto, se trabajó en que un usuario pudiese pinchar en la zona de subir foto, se abriese su galería y se mostrara en la interfaz la fotografía que había seleccionado.

Para ello, se utilizó el *plugin* nativo de cámara que ofrece *Capacitor* (y por ello, solo se puede probar en un *smartphone*). Lo que se hizo fue indicarle al *plugin* de *Capacitor* dónde buscar la foto (en la galería o hacer una foto en el momento), la calidad de la imagen y en qué formato se necesita, entre otros parámetros.

En concreto, se escogió que devolviese la imagen en Base64. Después, se transformó este resultado a una cadena de texto y se le añadieron los datos necesarios (el formato de la imagen, que estaba en Base64…) para generar una imagen legible por el HTML. Con esto, ya se podía elegir una foto desde un móvil y mostrarla en pantalla.

El siguiente paso fue, entonces, el de coger este contenido y subirlo al *Storage*. El formato en Base64 permitiría subir la imagen sin problemas, habiéndola transformado primero a *Blob*[38] mediante un algoritmo. Para subirla con *AngularFire*, se le ha de pasar el nombre que se quiere para la imagen (en este caso, la fecha actual y un ID aleatorio) y el *Blob* en sí, además de indicar el formato de imagen. Además, estas se almacenaron en una carpeta concreta para tener los archivos organizados.

Una vez subida al *Storage*, se guardaba la URL donde se encontraba. Al crear el viaje, esta se almacenaba en el campo de la foto de portada. Tras averiguar cómo seleccionar las fotos de la galería y subirlas a *Firebase*, las subidas de los iconos de perfil y de los archivos compartidos dentro de un viaje serían más fácil implementar en el futuro.

[38] *Binary large object*.

Tras terminar esta tarea, se continuó con la autenticación de usuarios. Por suerte, *Firebase Authentication* simplifica mucho este proceso y, además, almacena de forma segura las credenciales de los usuarios.

Antes de entrar en la funcionalidad, primero se debía tener el formulario de registro para recoger los datos. Así que se maquetó la interfaz de inicio de la aplicación [ID1], desde la cual se accede a la pantalla de registro [ID2] y la de inicio de sesión [ID3]. Para ello, se modificó el *routing* de la *app* y también se maquetaron estas dos últimas interfaces.

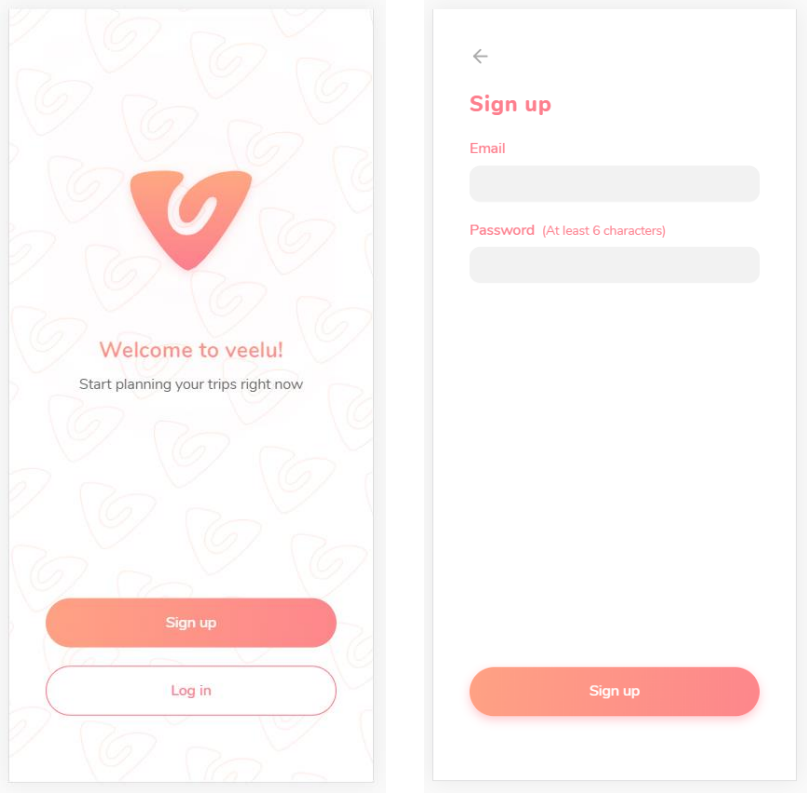


Figura 49. Interfaces de inicio y registro.
(Fuente propia)

Aunque *Firebase Authentication* también permite registrarse mediante proveedores como *Google, Facebook, Twitter*..., de momento se hizo de la forma más básica: con un e-mail (único) y una contraseña de más de 6 caracteres (según los requerimientos de *Firebase*).

Si el correo está repetido o la contraseña no es lo suficientemente larga, se muestra un modal avisando al usuario. Una vez se tenían estos datos, era tan fácil como utilizar una función de *AngularFire* para almacenarlos en *Firebase Authentication*.

Sin embargo, el problema de *Firebase Authentication* es que no permite gestionar los usuarios a no ser que se utilice el SDK de los administradores. Esto quiere decir que utilizando la *app* no se puede listar, por ejemplo, los usuarios que están registrados.

Para solucionar esto, se almacenó el e-mail y la contraseña en *Authentication* y se guardó el ID de usuario generado en el *Storage* del propio *Ionic*. De esta forma, se tenía acceso a él desde cualquier parte de la aplicación. Después, en la base de datos, en la colección "Usuarios", se creó un documento cuyo ID es el que estaba guardado y se almacenaba el *nickname* y el icono del usuario. Por defecto, el *nickname* es la primera parte del e-mail y el icono es una imagen predefinida.

De esta manera, el e-mail y la contraseña se hallaban de forma segura en *Firebase Authentication* y los datos genéricos, como el *nickname* y el icono, en la *Cloud Firestore*. Tras el registro o inicio de sesión, obtener el *nick* y el icono de la base de datos es sencillo, pues el documento en el que se encuentran tiene el mismo ID que el del usuario autenticado.

El siguiente paso fue añadir el usuario al mapa de participantes de un viaje al crearlo. Según el modelo de datos del capítulo anterior, se almacena su ID, rol, *nickname* y foto de perfil.

Con esa información ya almacenada, se pudo listar en la página principal los viajes de ese usuario en concreto. Para ello, con *AngularFire*, se indicó la colección ("viajes") y se hizo una consulta para que devolviese todos los viajes del usuario autenticado. En caso de no haber ningún viaje asociado, se muestra un mensaje animando al usuario a que cree uno nuevo.

9.5. Sprint 5: filtros y búsquedas de viajes y usuarios

Al llegar al 5º *sprint*, ya se había trabajado en profundidad con *Ionic*, *Angular* y todos los servicios proporcionados por *Firebase*.

El siguiente objetivo fue empezar con en el filtrado y la búsqueda de viajes. Tras investigar en Internet, se descubrió que, en el momento de realización de este TFG, *Firebase* está muy limitado en el tema de consultas a la base de datos.

Por ejemplo, si un usuario tiene entre sus viajes uno con el título “Viaje de fin de carrera” y quiere encontrarlo escribiendo la palabra “carrera”, *Firebase* no es capaz de realizar una consulta tan básica como esa.

Después de investigar cómo arreglar este problema, las únicas soluciones eran las siguientes: buscar una alternativa a *Firebase* o utilizar un motor de búsqueda para bases de datos NoSQL (como *Elasticsearch* [43] o *Algolia* [44]). Como es de esperar, la primera opción quedó descartada debido a lo avanzado que estaba el proyecto y a las grandes modificaciones que esto habría supuesto.

Por otro lado, utilizar un motor de búsqueda externo era la solución más propuesta por los desarrolladores que habían sufrido este mismo problema. Sin embargo, *Elasticsearch* no se puede utilizar a no ser que se disponga de un plan de pago en *Firebase* (ya que requiere el uso de *Google Cloud Platform*). En cuanto a *Algolia*, la solución recomendada por la propia web de *Firebase*, suponía duplicar toda la información de la base de datos a esta plataforma, traduciéndose en pagar mucho dinero.

Todos estos inconvenientes derivaron en la siguiente pregunta: ¿hasta qué punto son importante las búsquedas dentro de la aplicación? La realidad es que *veelu* solo tiene dos buscadores: uno para los viajes y otro para los usuarios. Por suerte, ofrecer un buscador completo no era algo vital para la *app*. Al final, se decidió que la mejor solución era proporcionar un sistema de búsqueda más limitado, pero que cumpliese su función básica.

Siguiendo los consejos de programadores que habían optado por la misma opción, se logró filtrar los viajes si se introducía el principio de su título. Volviendo al ejemplo anterior, ahora, al buscar “Viaje de”, al usuario le saldría en los resultados el titulado “Viaje de fin de carrera”. Por desgracia, si intentase buscarlo escribiendo “fin de carrera”, *Firebase* no sería capaz de encontrar dicho resultado.

Como se ha comentado antes, teniendo en cuenta que el buscador no es muy importante para la aplicación, se dejará con esta limitación. Si se tratase de una *app* con un enfoque diferente o de una escala mucho mayor, sí debería contemplarse cambiar a otro proveedor o utilizar los motores de búsqueda NoSQL.

Una vez implementada la consulta, se encontró otra limitación: *Firebase* hace distinción de caracteres en mayúscula y minúscula. Esto quiere decir que, si buscase "viaje de", el usuario no encontraría el viaje en cuestión, pues el título de este empieza con una mayúscula.

De nuevo, esta es otra gran restricción de *Firebase* y no se puede configurar. Resulta sorprendente que una plataforma del propio *Google* pueda estar tan limitada en aspectos tan básicos como estos. Por suerte, esto se puede solucionar creando otra variable en el documento del viaje que almacene el título, pero todo a minúsculas. Y después, al hacer una búsqueda, pasar todo el texto a minúsculas también. Así, al comparar ambos contenidos, *Firebase* devolvería el viaje que se busca, evitando el problema anterior.

Con el buscador ya funcionando, el siguiente paso fue trabajar en los filtros. Se decidió hacer unos botones que mostrasen los viajes clasificados según su fecha de inicio y fin, es decir, filtrar por los viajes ya pasados, mostrar solo los del futuro…

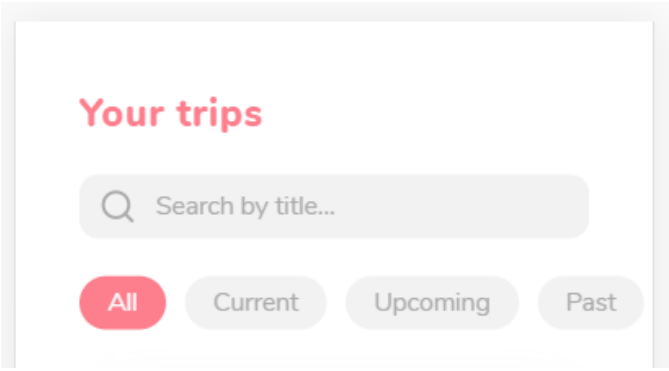


Figura 50. Buscador y filtros de los viajes.
(Fuente propia)

En principio, esto fue una tarea sencilla. Por defecto, se filtraba por el botón "*All*", así que se listaban todos los viajes sin más. Al seleccionar "*Past*" o "*Upcoming*", se hacía una consulta donde la fecha del fin del viaje fuese menor a la actual o donde la fecha inicial de este fuese mayor a la actual, respectivamente.

El problema se encontró al intentar filtrar los viajes activos en el momento actual (es decir, al filtrar por "*Current*"). En teoría, habría sido tan fácil como hacer una consulta donde la fecha inicial fuese menor o igual a la actual y, la fecha final, mayor o igual la actual.

Una vez más, *Firebase* no permite hacer una consulta tan sencilla como esta. En este caso, se debe a que no deja utilizar operadores (mayor, mayor o igual, igual…) en campos distintos en la misma consulta.

De nuevo, se investigó cómo poder abordar este problema. La solución más viable, y que es la que se adoptó, fue la de filtrar los viajes por uno de estos parámetros y después, en la parte del cliente, aplicar el siguiente filtro. A pesar de ser una solución poco óptima y negativa en cuanto a rendimiento, seguía siendo mejor que reemplazar *Firebase* por otra plataforma.

Así que, dentro de lo malo, se debatió acerca de cómo causar el menor impacto en el rendimiento de la aplicación. Para ver de forma clara cómo abordar el problema, es mejor pensar en el futuro, después de que los usuarios hayan utilizado la *app* durante un tiempo. Llegados a ese punto, la mayoría habrán acumulado muchos viajes pasados y lo más probable es que tengan solo un par de viajes futuros planeados.

Dicho de otra manera, imaginando el caso de un usuario que viaja con mucha frecuencia, lo más seguro es que, después de un año, tenga unos 4 o 5 viajes pasados y 1 o 2 pendientes de realizarse. Después de otro año igual, tendría acumulados un total 10 viajes pasados frente a 1 o 2 viajes para el futuro.

Por lo tanto, lo más inteligente fue que, al filtrar por "*Current*", *Firebase* devolviese todos los viajes cuya fecha de finalización fuera mayor o igual a la actual. Esto quiere decir que devolvería los viajes activos en ese momento más los viajes futuros. Con el paso del tiempo, los usuarios tendrán muchos más viajes pasados que futuros, así que de esta forma no se descargaría tanta información. De hecho, a no ser que un usuario planifique una gran cantidad de viajes para el futuro, esta solución no afectaría prácticamente al rendimiento.

Cuando se finalizó con el filtrado y la búsqueda de viajes, se comenzó con la maquetación de las interfaces de usuario. En este *sprint*, se maquetó el perfil de usuario [ID6], su página de editar los datos [ID7] y la de los ajustes [ID8]. Después, se crearon la del listado de amigos [ID9] y la que permite enviar solicitudes de amistad [ID10].

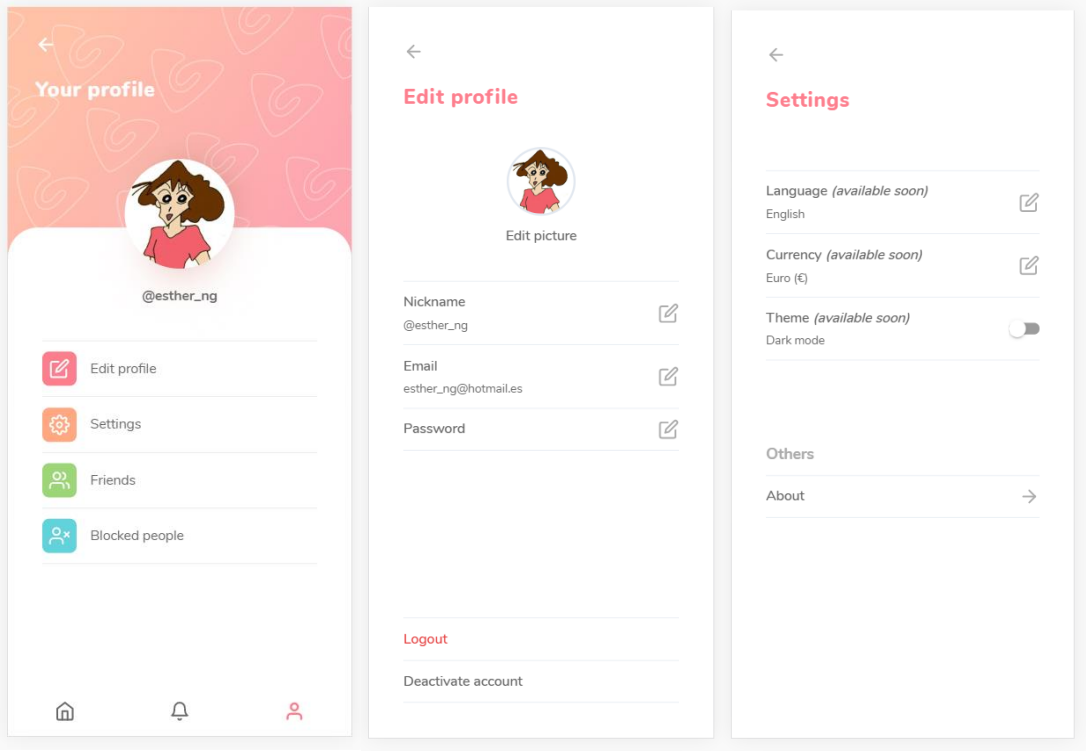


Figura 51. Interfaces de la sección de usuario.
(Fuente propia)

Por último, se implementó la búsqueda de usuarios. Al igual que con los viajes, se seguían teniendo ciertas limitaciones debido a *Firebase*. Si alguien quisiera buscar al usuario "Esnagar22" escribiendo "gar", no aparecería nada.

No obstante, lo normal es que un usuario busque a otro introduciendo su *nickname* exacto o la primera parte de este, ambos escenarios válidos para el buscador, así que el usuario no notaría ningún tipo de restricción.

Estos fueron todos los progresos del quinto *sprint*. Después de algo más de un mes de desarrollo, ya se disponía de una *app* sólida y con bastantes funcionalidades. Además, también se tenían implementadas bastantes interfaces. En la Figura 52, aparecen todas las pantallas, marcándose con un borde verde aquellas que estaban acabadas en este *sprint*:

Figura 52. Mapa de interfaces, destacando las implementadas.
(Fuente propia)

9.6. Sprint 6: peticiones de amistad y notificaciones push

Teniendo el buscador de usuarios implementado, para este *sprint* se establecieron como objetivos que un usuario pudiese agregar a otro y que, al enviarle la petición de amistad, el otro usuario recibiese una notificación en su móvil.

Al buscar usuarios, además de mostrar sus iconos y *nicknames*, es necesario que aparezca también un botón para seguir o dejar de seguir a dicho usuario. Para ello, se ha de conocer la relación entre el usuario obtenido en la búsqueda y el autenticado.

Entre dos usuarios se pueden dar tres situaciones: que no sean amigos, que sean amigos o que un usuario haya enviado la petición a otro, pero este no la haya aceptado aún. Si se trata de uno de los dos últimos escenarios, esta información estará almacenada en un documento de la base de datos con el estado de la petición de amistad (“aceptada” o “pendiente”).

Para averiguar cuál es el tipo de relación entre dos usuarios, lo más sencillo fue enviar el listado de amigos desde la interfaz de amigos a la interfaz para agregar uno nuevo. De esta forma, se evitó hacer peticiones innecesarias a la *Cloud Firestore*. Después, fue tan fácil como comprobar si los usuarios que coincidían con la búsqueda se encontraban en ese listado o no.

El problema al intentar obtener esta lista de amigos fue el siguiente: se tenía que hacer una petición que devolviese todos los documentos de “Usuarios_amigos” donde el valor de los campos “*nick1*” o “*nick2*” coincidiese con el del usuario que había iniciado sesión. Y *Firebase*, de nuevo, no permite hacer una petición tan básica como una condicional de tipo OR.

La solución a este inconveniente fue, entonces, intentar darle un enfoque diferente y hacer cambios en el diseño de la base de datos. Como resultado, se llegó a la conclusión de que con el operador “*array-contains*” de *Firebase* se podría evitar este problema.

La función de este operador es, a partir de un valor dado, comprobar si se encuentra dentro del *array* indicado (ignorando la posición exacta). Por lo tanto, creando un *array* que almacenara los *nicks* de los usuarios amigos (en lugar de ser campos separados), se pudo obtener el listado de amigos de un usuario. En la siguiente imagen se muestra el antes y el después de la estructura del documento que almacena esta información:

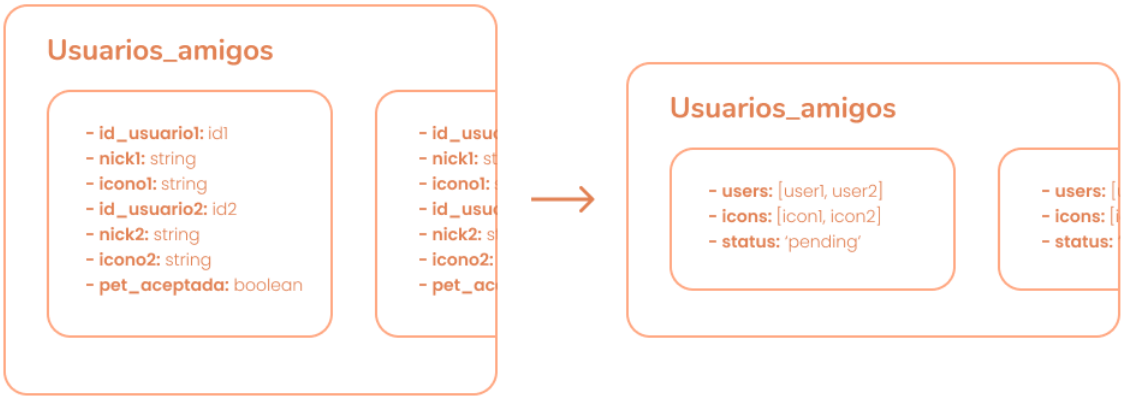


Figura 53. Cambios en el modelo de datos.
(Fuente propia)

Firebase no suele recomendar el uso de *arrays* por el tema de la sincronización de datos. Por ejemplo, si un usuario borra un elemento de un array mientras otro intenta leerlo, lo más probable es que salte un error. Por este motivo, en la mayoría de casos es mejor usar mapas.

No obstante, en esta situación concreta, el tamaño del *array* nunca cambia, ni siquiera cuando dos usuarios dejan de ser amigos (pues se elimina el documento directamente). Así que, para este caso, no hay ningún riesgo en utilizar *arrays*.

Después de obtener los datos de la petición de usuarios amigos, se crearon componentes reutilizables, como el formado por el icono y el *nickname* de un usuario o el del botón para seguir o dejar de seguir a alguien.

Una vez se disponía de los componentes, se implementaron las interfaces de usuarios amigos [ID9] y la de agregar uno nuevo [ID10]. Al hacer una búsqueda, se iba recorriendo cada uno de los usuarios resultantes y se iban añadiendo a un *array* el icono del usuario, el *nickname* y la relación con el usuario en cuestión. Después, se implementó la funcionalidad para seguir o dejar de seguir a un usuario. Al darle al botón "*follow*", este pasa a ser "*pending*" hasta que el otro usuario acepte la petición.

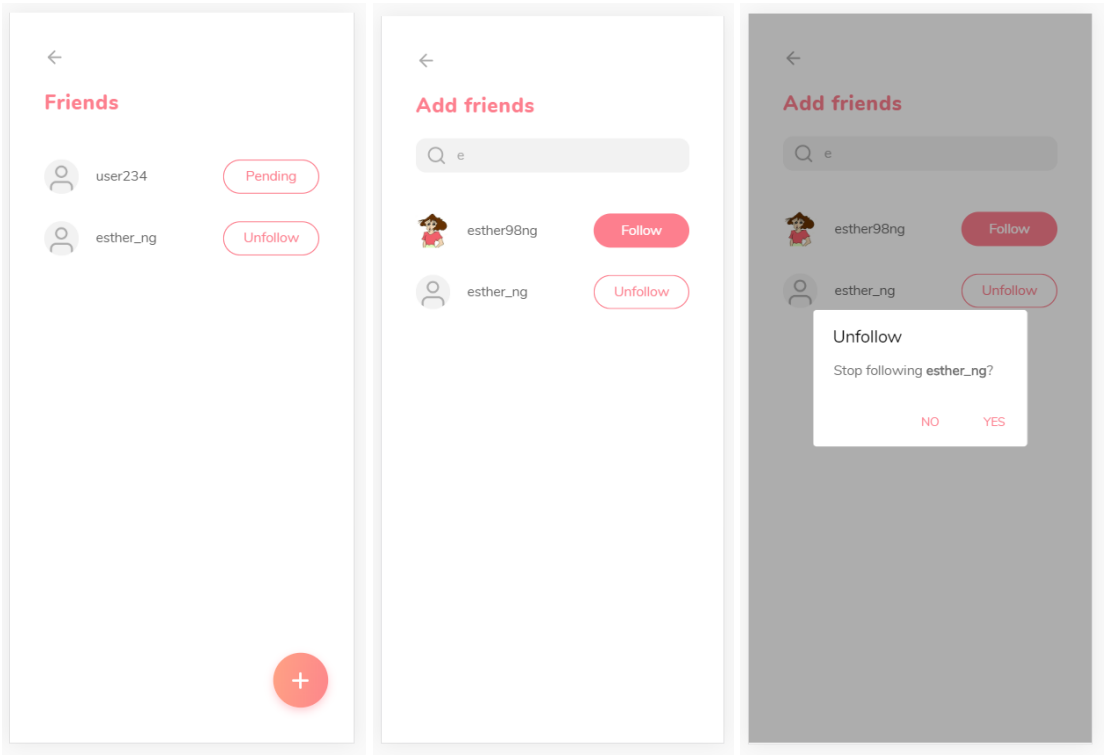


Figura 54. Interfaces de usuarios amigos y agregar amigo.
(Fuente propia)

Como se muestra en la última de las capturas, en el caso de pulsar el botón "*unfollow*" o "*pending*" (es decir, se quiere dejar de ser amigos con dicha persona), se muestra un modal de confirmación de la acción. De esta forma, se mejora la experiencia de usuario al evitar que una persona deje de seguir a otra accidentalmente.

El siguiente paso fue implementar la interfaz de notificaciones [ID5]. Desde esta pantalla, se obtienen todas las peticiones de amistad de un usuario donde su estado es "*pending*". Para evitar que saliesen las solicitudes enviadas por el propio usuario, se filtraban en la parte del cliente teniendo en cuenta que, tanto en el *array* de *nicks* como en el de iconos, los datos de quien manda la petición están en la primera posición.

Por último, se desarrolló la funcionalidad para aceptar o rechazar la petición. Con todos estos avances, los usuarios ya eran capaces de agregarse entre sí.

Una vez terminadas las peticiones, se pasó a lanzar notificaciones *push* al móvil tras recibir una solicitud de amistad. Para ello, *Capacitor* ofrece varios *plugins* que implementan esta funcionalidad nativa. En su documentación oficial incluyen un tutorial para configurar las notificaciones para *Firebase* [45].

El primer paso para hacer funcionar las notificaciones es registrar el dispositivo móvil que se está utilizando. En el siguiente esquema, se puede ver el funcionamiento interno de *Capacitor* para generar un *token* único y registrarlo:

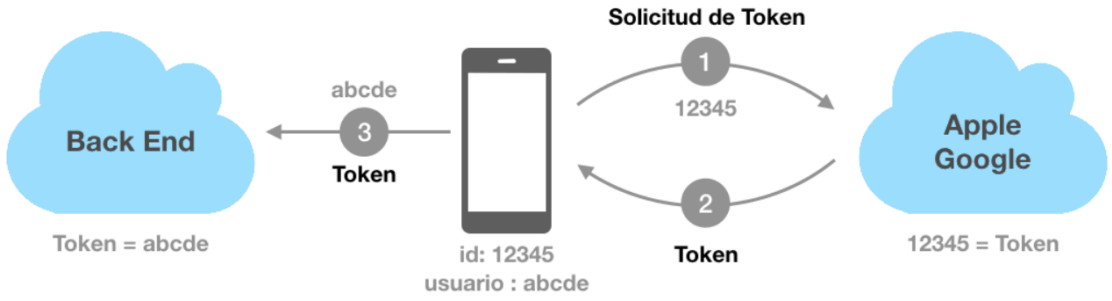


Figura 55. Flujo de obtención de tokens de dispositivos con Capacitor.
(Fuente: https://medium.com/@brankofuenzalida/entendiendo-el-plugin-push-notifications-de-capacitor-8ca84cdd8d38)

En primer lugar, *Capacitor* hace una solicitud a *Apple* (en caso de *iOS*) o *Google* (si se trata de *Android*) para obtener un *token* único que identifique el dispositivo. Una vez se dispone de dicho *token*, es trabajo del desarrollador almacenarlo en su base de datos, asociándolo a un usuario. En este caso, se creó una nueva colección en la *Cloud Firestore* llamada "*devices*", encargada de almacenar el valor de un *token* junto al *nickname* de su usuario.

Por otro lado, los *plugins* de *Capacitor* son capaces de detectar cuándo se han producido ciertos eventos [46]. Estos eventos son "*registration*", "*registrationError*", "*pushNotificationReceived*" y "*pushNotificationActionPerformed*".

Los dos primeros, como su nombre indica, se activan cuando el registro sale bien o cuando hay algún error en el proceso. Si se registra el dispositivo y se obtiene el *token* correctamente, entonces se envía a *Firestore* una petición para guardar los datos.

Por otro lado, los dos últimos eventos se activan al recibir la notificación en sí. En concreto, "*pushNotificationReceived*" se emite cuando está la *app* abierta y se recibe una notificación. En el caso de *veelu*, en ese momento se cambia el icono de la campana de las notificaciones por otra campana con pequeño círculo. Por su parte, "*pushNotificationActionPerformed*" se activa cuando se pincha la notificación *push* que llega al teléfono. Cuando se da este evento, se redirecciona al usuario a la página de notificaciones.

Tras entender el funcionamiento de las notificaciones *push* de *Capacitor*, fue el momento de configurar *Firebase* de forma que se pudieran lanzar las notificaciones desde el *backend* [47].

En primer lugar, desde la consola de *Firebase*, se creó una nueva aplicación *Android*. Aunque *Capacitor* lo permite, no se han implementado las notificaciones para *iOS* puesto que no se dispone de ningún dispositivo para hacer pruebas.

Para generar una aplicación *Android* en *Firebase*, hay que introducir el identificador de la *app*. Este *ID* se puede encontrar en el archivo de configuración de *Capacitor* y tiene que seguir la estructura "*com.company.appname*". Al terminar, se genera el archivo *google-services.json*, que contiene toda la información que *Capacitor* necesita y que se ha de añadir en la carpeta raíz de la aplicación *Android* del proyecto.

A continuación, para poder detectar que se había enviado una petición de amistad, fue necesario utilizar las llamadas *Cloud Functions*. Estas son otro de los servicios que ofrece *Firebase*, el cual no había hecho falta hasta ahora.

Cloud Functions es un *framework* sin servidores que permite ejecutar automáticamente código *backend* al recibir solicitudes HTTPS o al detectar eventos de *Firebase*. Para ello, se han de utilizar funciones *JavaScript* o *TypeScript* en un entorno *Node.js*.

El primer paso, entonces, fue iniciar *Cloud Functions* dentro del proyecto e instalar las dependencias. En el archivo principal, se importaron los paquetes necesarios: el de *Cloud Functions* y el SDK de administrador de *Firebase*, necesario para enviar las notificaciones.

A continuación, se creó una función que hace lo siguiente: al detectar la creación de un nuevo documento en la colección "Usuarios_amigos", coge los datos y guarda los *nicknames* de los usuarios implicados. Luego, crea un JSON con el mensaje que recibirá el usuario. Seguidamente, hace una petición a la base de datos para obtener el *token* de dispositivo del usuario al que le envían la solicitud, pasándole su *nickname*. Por último, con el SDK de administración de *Firebase*, se le envía a dicho dispositivo el mensaje creado.

Una vez implementada la función, se hizo *deploy*[39] a *Firebase* y se pudo comprobar cómo llegaban notificaciones a un móvil al recibir una solicitud de amistad.

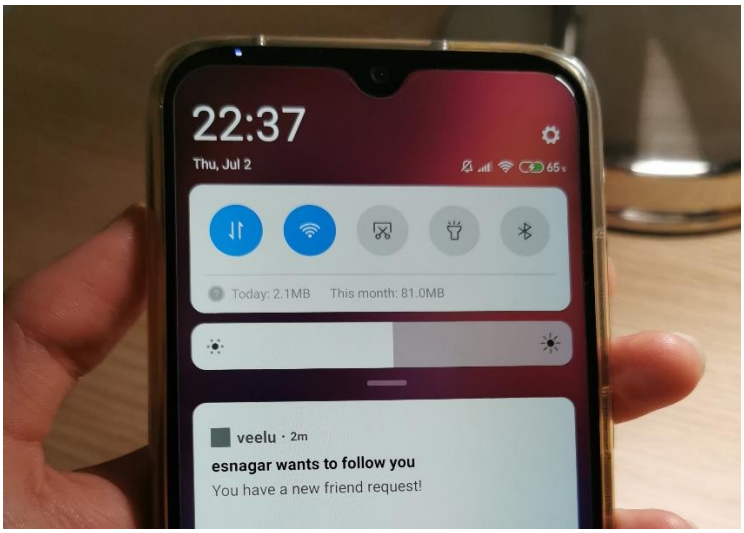


Figura 56. Notificación push de una solicitud de amistad en un Xiaomi Redmi Note 8. (Fuente propia)

[39] Desplegar, subir a producción.

9.7. Sprint 7: gestión colaborativa de viajes y herramientas

Al llegar al séptimo *sprint* del desarrollo de *veelu*, hubo que replantearse el nivel de prioridad de las tareas que no se habían realizado todavía y cuáles se debían descartar. Esto se debe a que, con la fecha límite de entrega de este TFG cada vez más cerca, no se disponía del tiempo suficiente para implementar toda la aplicación.

Por lo tanto, se revisaron las tareas del *backlog* del tablero y se ordenaron de mayor a menor prioridad, como se muestra en la siguiente captura:

Figura 57. Actualización del Backlog en el Sprint 7.
(Fuente propia)

Se decidió que lo más importante era implementar lo relacionado con añadir o quitar los usuarios que participan en un viaje, puesto que la gestión colaborativa es una de las características esenciales de *veelu*. Para ello, también se necesitaba programar la edición de un viaje y así demostrar que se veían los cambios para el resto de usuarios.

Después de estas tareas, se dio prioridad a aquellas relacionadas con las herramientas de un viaje: las actividades, el diario, el *checklist*, etc. Esta decisión se tomó teniendo en cuenta el orden

de prioridad que se estableció al diseñar las interfaces. Como el objetivo era que *veelu* fuese una aplicación todo en uno, las herramientas de un viaje eran tareas preferentes.

No obstante, para cada una de estas herramientas, habría que maquetar sus interfaces de listado, detalle, creación, edición y borrado, así como implementar la funcionalidad de cada una de estas acciones. Todo este trabajo requiere varias semanas de desarrollo de las que no se disponían, por lo que se le dio prioridad a la maquetación de las interfaces principales frente a la funcionalidad de las mismas.

En cuanto al resto de tareas, se encontraban algunas como la gestión de usuarios bloqueados, editar el perfil, los ajustes de la aplicación, añadir animaciones… Estas tareas, aunque son necesarias para conseguir un producto final completo y pulido, se consideraron más prescindibles y descartaron, de nuevo, por la limitación temporal.

Comenzando con el *sprint* en sí, se pasó a implementar la gestión de usuarios de un viaje. No obstante, al poco de comenzar con esta tarea, fue necesario rediseñar parte de la base de datos. Hasta ese momento, según la estructura definida en el apartado de diseño, los participantes se almacenaban dentro de un viaje de la siguiente forma:

Figura 58. Antigua estructura de un documento de viaje.
(Fuente propia)

Las claves "*user_1*" y "*user_2*" son ejemplos de *nicks*. Aunque pueda parecer una estructura adecuada, el problema se encuentra al intentar averiguar si un usuario pertenece a un viaje: no se puede hacer una consulta que compruebe si, dentro de un mapa, existe una clave concreta.

Por lo tanto, se buscó una forma de solventar este problema. De nuevo, la solución fue hacer uso de *arrays*. En concreto, dentro de un "viaje", se creó un *array* llamado "idsParticipantes", encargado de almacenar los IDs de los usuarios pertenecientes al mismo. Además, para

depositar los datos de los usuarios en el mapa “participantes”, se comenzó a utilizar sus IDs como clave (en lugar de sus *nicknames*).

De esta forma, con el operador “*array-contains*” de *Firebase*, se podía averiguar si un usuario estaba asociado a un viaje concreto al comprobar si su ID aparecía en “idsParticipantes”. Si fuera este el caso, se accedería al resto de información guardada en “participantes”. En la siguiente captura de *Firebase*, se puede ver la nueva estructura de estos datos:

Figura 59. Estado actual de la estructura que almacena los participantes de un viaje. (Fuente propia)

Este nuevo diseño, a su vez, provocó cambios en los documentos pertenecientes a la colección de “Usuarios_amigos”, ya que hasta el momento no se almacenaban los IDs de los usuarios implicados. En esa situación, un usuario no podría añadir a un amigo a un viaje (pues hace falta conocer su ID). Por este motivo, se creó un *array* para cada documento de “Usuarios_amigos” que guardase los identificadores de ambos.

Tras superar este contratiempo, se pasó a la maquetación de la interfaz de participantes [ID15]. Como se comentó en el apartado de gestión de contenidos, solo el administrador es capaz de gestionar los participantes. Si un usuario normal entra a la interfaz, solo se muestran los usuarios asociados del viaje. Si se trata del administrador, se listan todos sus amigos, permitiéndole añadirlos o quitarlos.

Para ello, el administrador tiene que marcar o desmarcar el botón de tipo *check* asociado a cada usuario. Los participantes del viaje se muestran en la parte superior de la interfaz. Para confirmar los cambios, hay que pulsar el botón flotante.

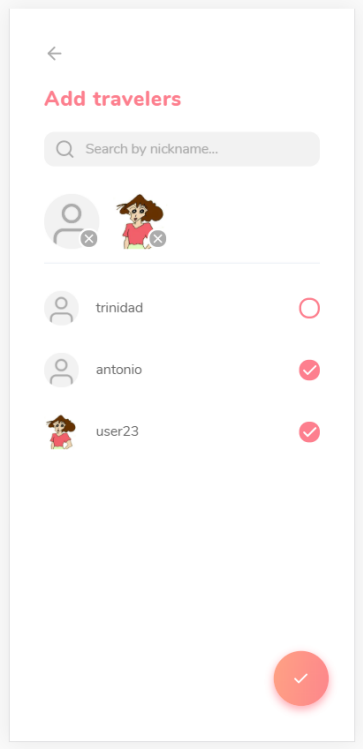


Figura 60. Interfaz de gestión de participantes de un viaje.
(Fuente propia)

Uno de los objetivos al implementar la gestión de participantes fue que, tanto la interfaz como el funcionamiento, fuesen similares a cuando se crea un grupo en *WhatsApp* o *Telegram*. De esta manera, el proceso es más intuitivo para el usuario.

Después de la gestión de usuarios, se pasó a la edición de un viaje. Gracias al componente de *Ionic* "*ion-popover*", se creó un *popover*[40] para mostrar las opciones de "editar" y "eliminar" de un viaje, al cual se accede pinchado en el icono de la parte superior derecha.

Una vez modificado el *routing*, para editar el viaje se reutilizó la interfaz del formulario para crear uno, con la diferencia de que, esta vez, se rellenaban los campos con la información ya existente. Después, se implementó la funcionalidad que hiciese persistentes los cambios. Utilizando dos dispositivos móviles, se comprobó cómo un usuario podía ver los cambios que otro había realizado sobre un viaje de forma instantánea.

[40] Cuadro de diálogo. Normalmente se utiliza para mostrar acciones que no caben en la interfaz o en la barra de navegación.

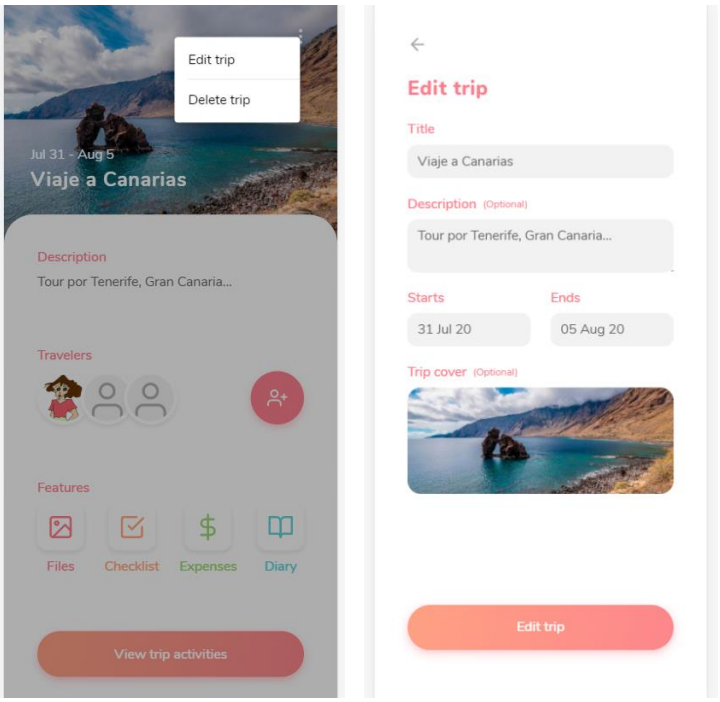


Figura 61. Interfaces de viaje (con popover) y edición de viaje.
(Fuente propia)

Por último, se generaron las interfaces del listado de actividades [ID16] y la de su detalle [ID19]. Como la maquetación tenía más prioridad que la funcionalidad, se creó información local de ejemplo, lo cual supuso que el proceso de maquetación se agilizase notablemente.

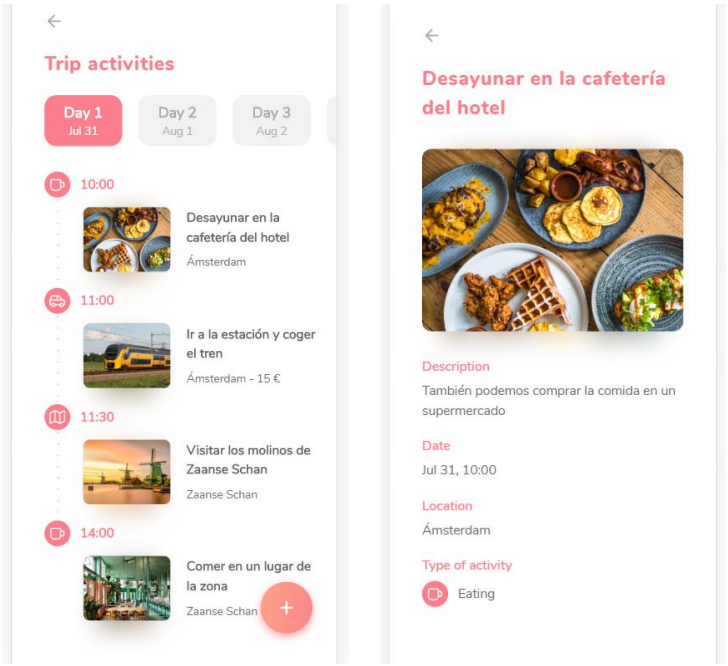


Figura 62. Interfaces de listado de actividades y detalle de actividad.
(Fuente propia)

9.8. Sprint 8: resto de herramientas, últimos arreglos y build

El *sprint* número 8 fue la última semana dedicada al desarrollo de la *app*. El objetivo de este *sprint* fue terminar las tareas pendientes y pulir la aplicación lo máximo posible.

En primer lugar, se continuó con la implementación de las herramientas de un viaje. Para el *checklist* [ID27], se creó un componente que sirviese para filtrar por ítems grupales o ítems individuales. Después, se maquetó el resto de la interfaz:

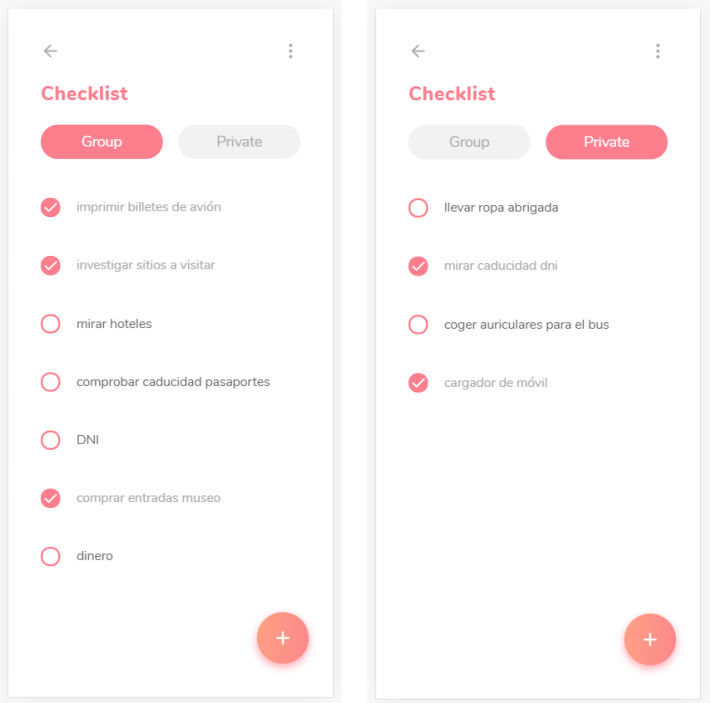


Figura 63. Ítems de grupo e ítems privados.
(Fuente propia)

La siguiente herramienta a maquetar fue la del diario. Tras crear información de ejemplo, se implementó la interfaz del listado de entradas [ID24]. A la hora de mostrar el texto de la entrada, se utilizó un *pipe* de *Angular* para limitar el número de caracteres.

Después, se modificó el *routing* de la *app* para que, al pinchar en una de las entradas, redirigiese al detalle de la misma [ID25]. Desde esta interfaz, además de mostrar los datos que ya aparecen en el listado, se puede leer todo el texto que contiene la entrada.

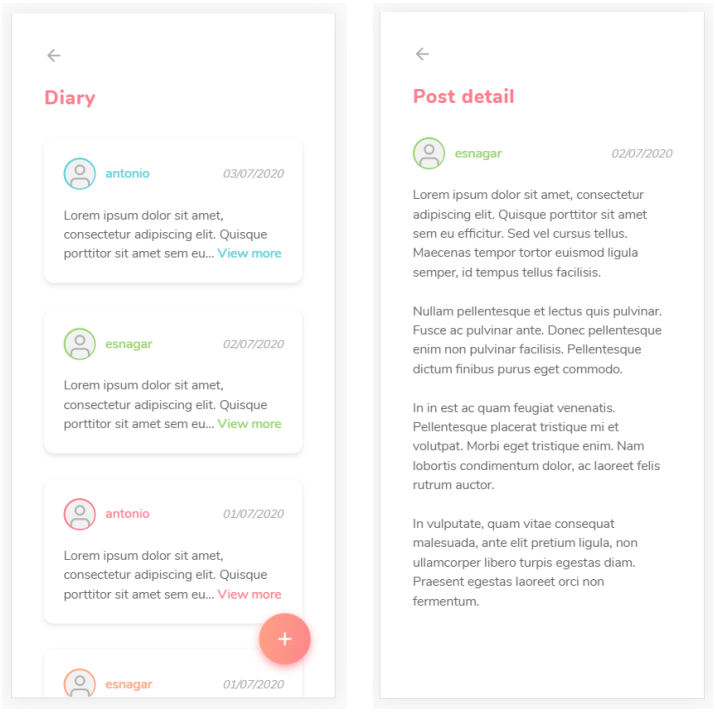


Figura 64. Entradas de diario y detalle de entrada.
(Fuente propia)

A continuación, se pasó a maquetar la interfaz de gastos del viaje [ID29] (diferenciando gastos grupales de individuales), así como el detalle al pinchar en uno de ellos [ID30].

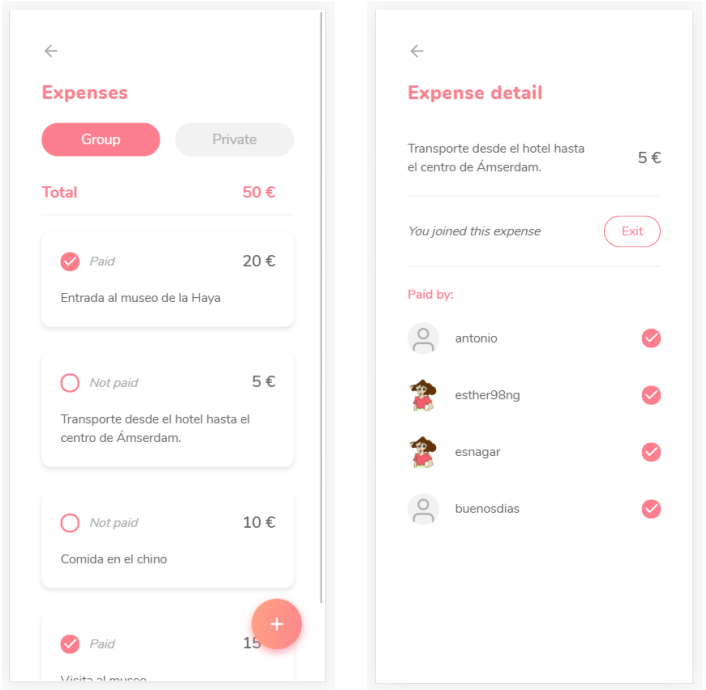


Figura 65. Interfaces de gastos de un viaje.
(Fuente propia)

Por último, se programó la sección de archivos de un viaje. Para ello, se distinguió entre carpetas y los archivos en sí. Gracias a los componentes predefinidos de *Ionic*, se pudo hacer un botón flotante que mostrara varias acciones (crear carpeta o subir archivo).

Además, se utilizó un plugin de *Cordova* (ya que con *Capacitor* se pueden usar tanto *plugins* nativos de *Cordova* como de *Ionic*) llamado *PhotoViewer*, que permite previsualizar las fotografías desde el dispositivo móvil.

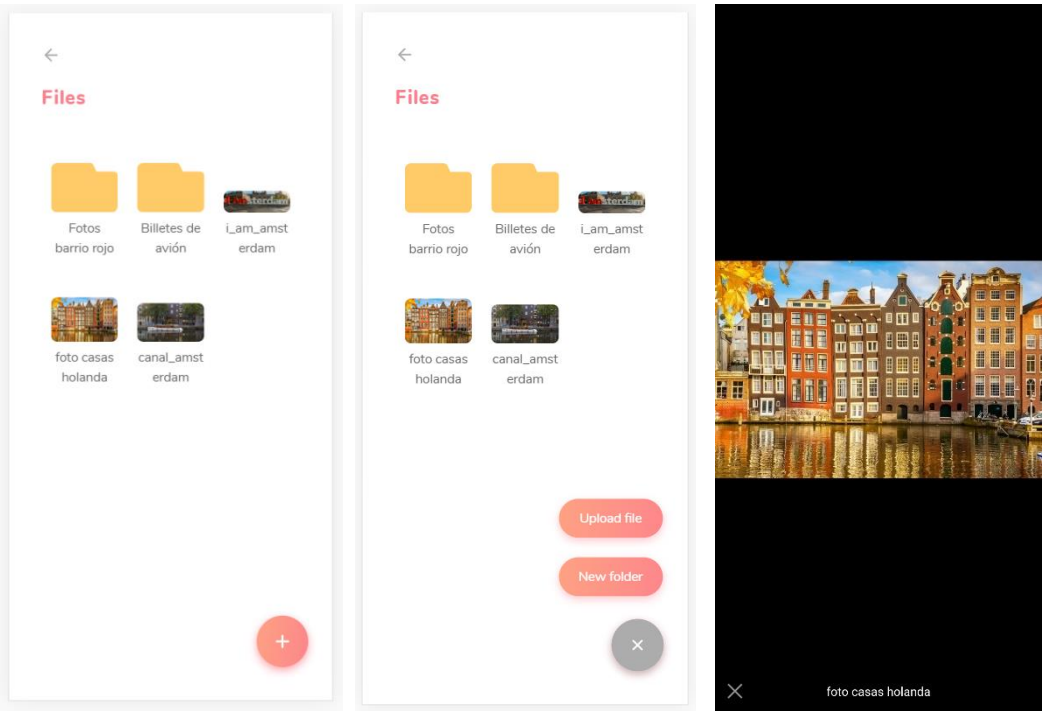


Figura 66. Interfaces de archivos de un viaje.
(Fuente propia)

Con estas últimas interfaces maquetadas, se dio por finalizado el desarrollo de *veelu*. A partir de este punto, no se añadieron más interfaces o funcionalidades nuevas, sino que se pasó a pulir la aplicación y a arreglar fallos.

En primer lugar, se mejoró el acabado de la *app*. Por ejemplo, se añadieron propiedades de CSS para modificar los estilos al pulsar un botón, enlaces, etc. También se cambió la estética de los campos de texto al situarse sobre ellos y se trabajó en mejorar el acabado de los modales, entre otras tareas.

Por último, se añadieron los iconos e imágenes *splash*[41] propios de *veelu*, sustituyendo a los que *Capacitor* añade por defecto.

Tras acabar con el apartado estético, se intentó mejorar el rendimiento de la aplicación. En concreto, el principal objetivo fue reducir los tiempos de carga al iniciar la *app*.

Según un estudio realizado por Google en 2015 [48], el 29% de usuarios cambian al instante de página web o *app* si esta no satisface sus necesidades. En concreto, el 70% de estos usuarios lo hacen debido a la lentitud de los tiempos de carga. Dos años más tarde, en otro estudio de Google [49], se descubrió que, si una página web tarda en cargar entre 1 y 5 segundos, la probabilidad de que el usuario cambie a otro sitio web se incrementa un 90%.

En definitiva, los usuarios están acostumbrados a que las páginas y aplicaciones móviles carguen de forma inmediata. Para evitar perder posibles usuarios, era importante que *veelu* tuviese un buen rendimiento y que el tiempo de carga de la *app* fuera el menor posible.

Hasta el momento, todas las páginas de *veelu* se cargaban mediante *lazy loading*[42]. Esta técnica consiste en descargar solo el HTML, CSS y *JavaScript* necesario para renderizar la primera ruta y el resto de partes de la aplicación se van cargando según se van necesitando.

La técnica de *lazy loading* es la que utiliza por defecto *Angular*. Además, al declarar las rutas de la aplicación, *Ionic* permite elegir una estrategia de *pre-loading*: hacer *pre-loading* de todos los módulos (*PreloadAllModules*) o de ninguno (*NoPreloading*).

Aunque la primera estrategia funcione bien para algunas aplicaciones, puede ralentizar la ejecución si se trata de una *app* extensa. Por este motivo, lo mejor es definir una estrategia intermedia, que haga *pre-loading* aquellos módulos que se van a necesitar más pronto. En el caso de *veelu*, era conveniente tener preparadas las interfaces de introducción [ID1], registro [ID2] e inicio de sesión [ID3].

Para ello, se investigó cómo poder establecer una estrategia sencilla que cargase las páginas especificadas [50]. En primer lugar, se definieron los módulos a cargar mediante un parámetro en cada ruta. Después, se sobrescribió la clase abstracta "*PreloadingStrategy*" (que proporciona *Angular*) para hacer *pre-loading* de las rutas que tuviesen el parámetro comentado

[41] Imagen de bienvenida que aparece mientras se carga la *app*.
[42] Carga diferida.

anteriormente. Por último, se estableció esta clase en *Ionic* como estrategia a seguir para cargar los módulos.

De esta forma, se combina el *lazy loading* con páginas ya cargadas. Este cambio mejoró la velocidad a la que se iniciaba la aplicación.

Por otro lado, se hicieron modificaciones para ocultar la pantalla de *Splash* de forma manual. Esto se debe a que *Capacitor*, por defecto, muestra durante 3 segundos esta imagen. Tras modificar este comportamiento automático y ocultando la pantalla de forma manual, se logró que la *app* se lanzase todavía más rápido.

Con la aplicación completamente desarrollada, pulida y optimizada, se pasó a generar el APK[43] de *veelu*. En primer lugar, se compiló el proyecto para producción, lo cual minimiza y optimiza el código e ignora *plugins* que están incluidos pero que no se utilizan. Después, tras abrir el proyecto en *Android Studio*, se generó el APK y, tras probarlo en varios dispositivos móviles, se dio por finalizado el desarrollo de *veelu*.

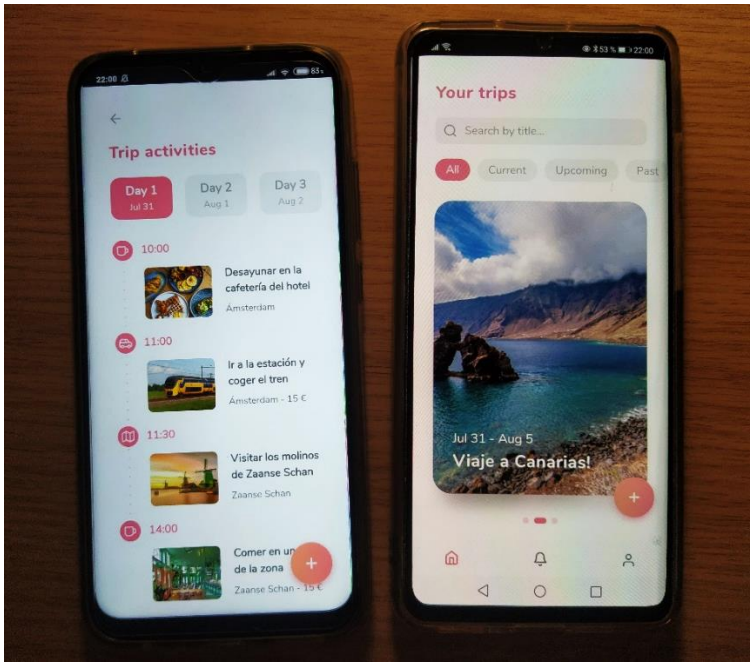


Figura 67. App final funcionando en un Xiaomi Redmi Note 8 y un Huawei P30 PRO. (Fuente propia)

[43] *Android Aplication Package*. Paquete ejecutable para el sistema operativo *Android*.

10. Pruebas y validación

En este apartado se llevan a cabo las pruebas y validación definidas en el capítulo de diseño. De esta forma, se comprueba si la aplicación está libre de errores y cumple su propósito o, si es el caso contrario, encontrar qué está fallando y solucionarlo.

Antes de comenzar con las pruebas, hay que aclarar que *veelu* no está finalizada al completo. Debido a lo extensa que es la *app* y al límite temporal del TFG, no se ha podido desarrollar a tiempo todo lo que se tenía pensado para la aplicación. Por este motivo, la comprobación de si se han cumplido los requisitos (funcionales y no funcionales) se llevará a cabo cuando la *app* se termine de implementar.

Además, al no estar lo suficientemente desarrollada, por el momento no podrá ser probada por usuarios reales. Estas opiniones se recogerán mediante el cuestionario de *Google Forms* cuando se disponga de una versión *beta*[44] de *veelu*.

Por otro lado, sí se probó la aplicación con los servicios que ofrece *Firebase*. Aunque la *app* no esté implementada del todo, es una buena práctica hacer *testing* durante el desarrollo para encontrar posibles errores y darles solución cuanto antes.

En primer lugar, subiendo el APK generado, *Firebase Test Lab* permitió ejecutar *veelu* en varios dispositivos (tanto físicos como virtuales). Se realizaron *tests* tipo "*Robo*", los cuales son inteligentes y automáticos, que probaron la funcionalidad de la aplicación. Para ello, fue necesario especificar unas credenciales válidas para que iniciara sesión y pudiese acceder a todo el contenido de la *app*. Se realizaron varias pruebas que se ejecutaron correctamente y no revelaron ningún problema.

En la Figura 68, se muestra el gráfico de rastreo de interfaces que ha generado una de las pruebas de *Robo*. En este caso, ha navegado por las pantallas de inicio de sesión, viajes, detalle de un viaje... hasta llegar a las actividades del mismo:

[44] Aplicación que se encuentra todavía en la fase de desarrollo y no en la versión final.

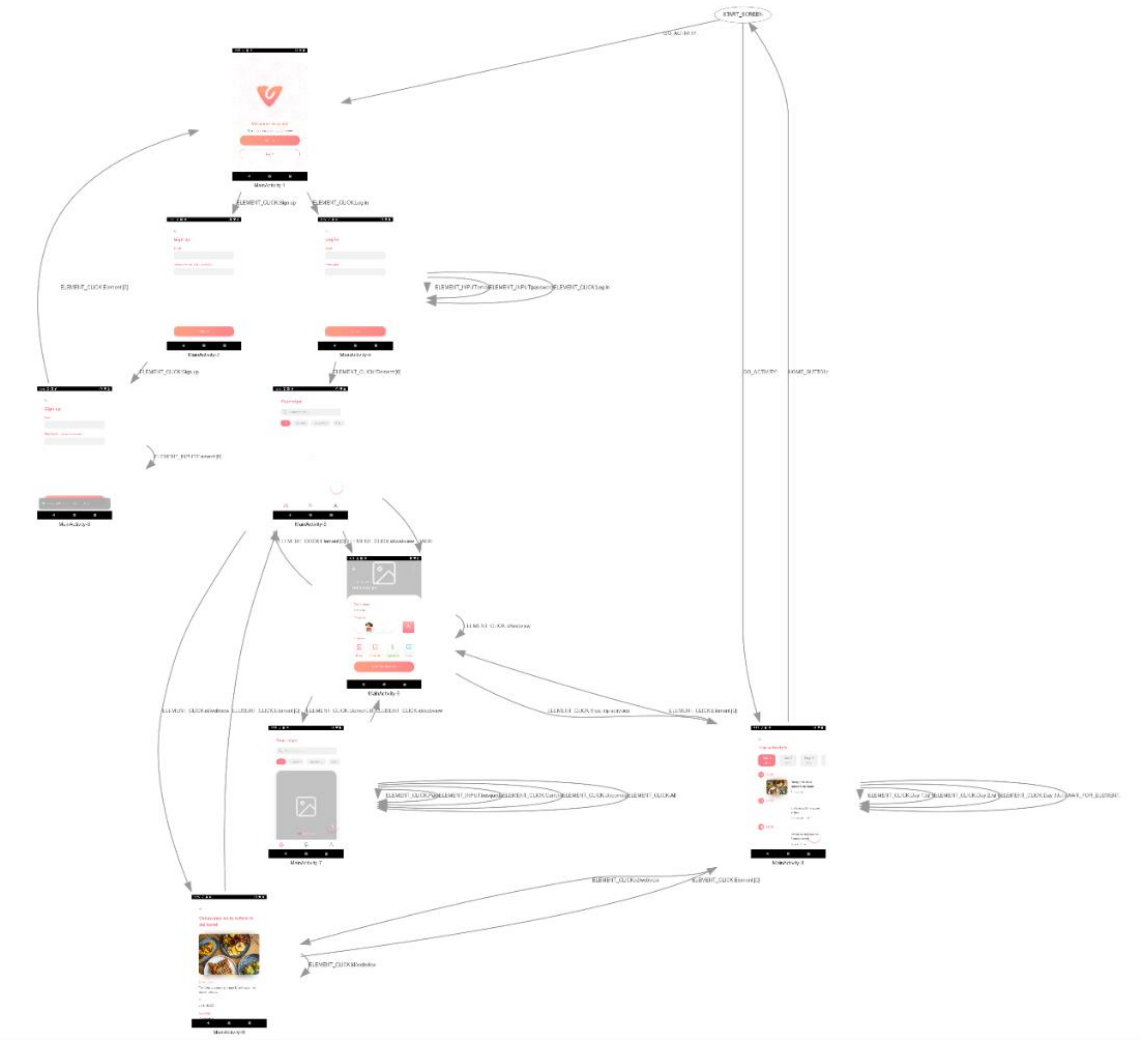


Figura 68. Gráfico de rastreo de una prueba con Firebase Test Lab.
(Fuente propia)

Después, tras haber añadido sus dependencias al proyecto, se pudo hacer uso de *Firebase Crashlytics* y *Firebase Performance Monitoring*.

En primer lugar, se probó la *app* desde los tres *smartphones* de los que se disponía: un *Xiaomi Redmi Note 8*, un *Xiaomi Mi A3* y un *Huawei P30 PRO*. A simple vista, en ninguno de estos dispositivos se produjeron errores y se pudo hacer uso de la aplicación sin ningún problema.

Desde *Crashlytics*, que se encarga de recoger fallos en tiempo real, tampoco se detectó ningún problema que pudiese afectar al funcionamiento de *veelu*. Por su parte, con *Firebase Performance Monitoring* se comprobó el correcto rendimiento de la aplicación.

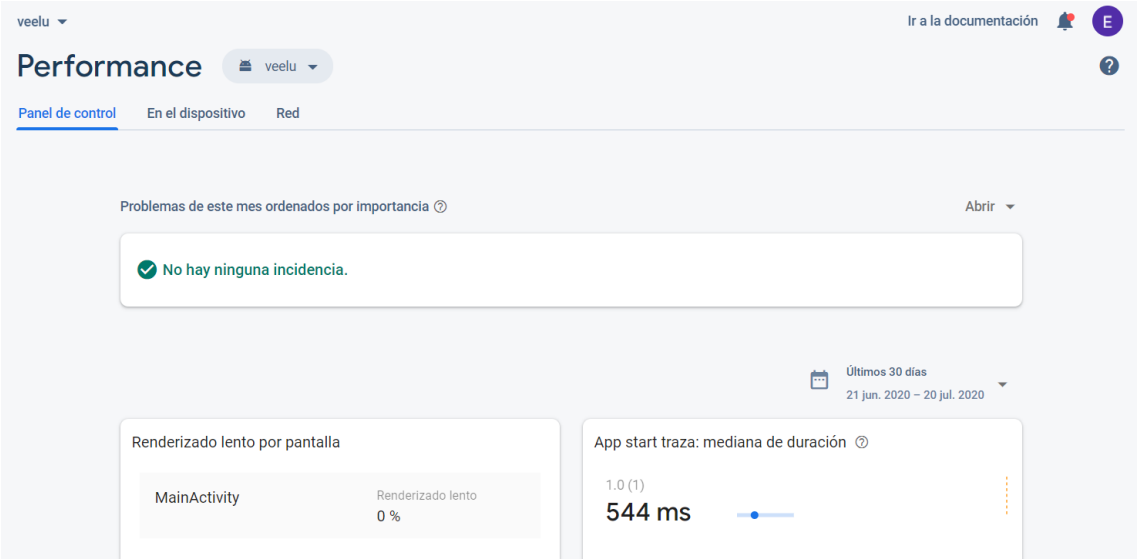


Figura 69. Panel de control de Firebase Performance.
(Fuente propia)

A pesar de que los resultados obtenidos han sido buenos, será necesario continuar con el proceso de *testing* una vez se haya desarrollado del todo la aplicación. En ese momento, cuando se pueda probar la *app* desde numerosos y variados dispositivos, se obtendrán resultados mucho más representativos y realistas que permitirán mejorar la aplicación.

11. Resultados

11.1. Producto final

La aplicación final se ha ido mostrando durante el apartado del desarrollo. En cuanto al total de interfaces que componen la *app*, a las diseñadas en *Figma* hay que sumarles las interfaces de edición, siendo un total de 36 pantallas. De ese número de interfaces, se ha logrado maquetar 23, que se traduce en el 62'8% de interfaces de *veelu*. En la siguiente figura, se destacan las interfaces maquetadas con un borde verde:

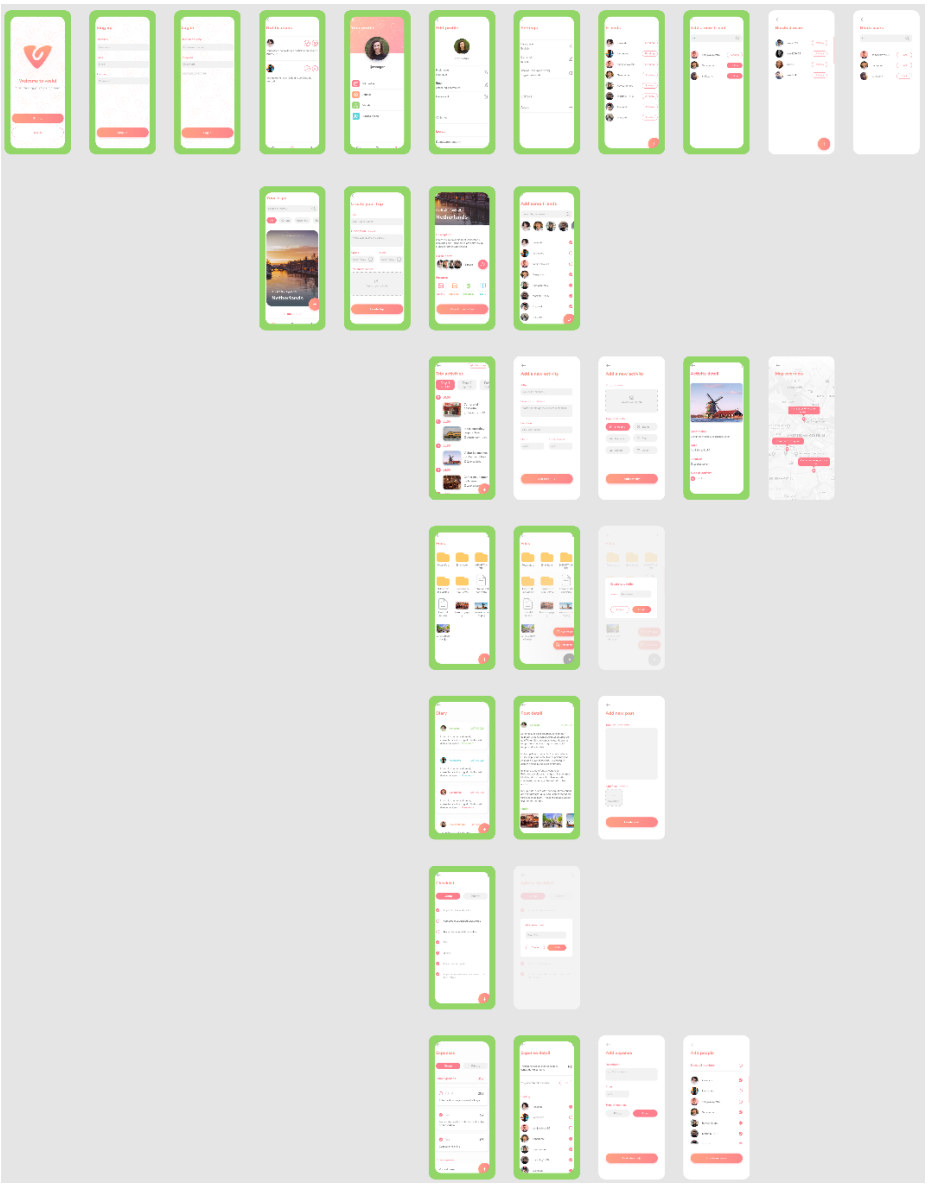


Figura 70. Interfaces implementadas.
(Fuente propia)

En cuanto a la funcionalidad de la aplicación, ha faltado implementar la gestión de las herramientas (actividades del viaje, *checklist*…), el perfil de usuario y los usuarios bloqueados, puesto que se dio prioridad a la maquetación frente a la funcionalidad.

Para hacer cálculos, se ha tenido en cuenta que, aproximadamente, cada interfaz cubre una funcionalidad concreta. Por lo tanto, se podría decir que se ha desarrollado el 44,4% de la funcionalidad de la *app* (16 interfaces funcionales frente a las 36 totales).

A pesar de no haber implementado la *app* al completo, siguen siendo unos buenos porcentajes de trabajo realizado si se tiene en cuenta la extensión de la aplicación. En definitiva, se necesitan varios meses más de desarrollo para que *veelu* esté terminada. No obstante, en este punto, se ha conseguido un MVP[45] que incluye las partes más importantes de la aplicación.

A la hora de realizar la defensa de este TFG, además de una introducción y conclusiones, se hará una demostración de la aplicación. Esta demo se llevará a cabo con dos dispositivos *Android*.

En primer lugar, con uno de estos móviles se mostrará la pantalla de inicio y se procederá a iniciar sesión con una cuenta ya existente. A continuación, se mostrará el listado de viajes del usuario y cómo pueden filtrarse, tanto por su fecha como a través del buscador.

El siguiente paso será, entonces, mostrar la sección del perfil de un usuario. Para ello, se irá navegando por sus diferentes subsecciones hasta llegar a la de usuarios amigos. Al acceder a esta interfaz, se hará una búsqueda para enviarle una petición al usuario que, previamente, habrá iniciado sesión desde otro *smartphone*.

En ese momento, aparecerá una notificación *push* en el otro dispositivo informando de la solicitud de amistad. Tras aceptarla, desde el primer móvil se accederá al detalle de un viaje y se añadirá a este nuevo usuario como participante. Después, se editará dicho viaje para que se observen los cambios instantáneos desde el otro móvil.

Por último, se hará un recorrido por todas las herramientas de un viaje, mostrando los listados y detalles de las mismas.

[45] *Minimum Viable Product*. Producto con las suficientes características para satisfacer a los clientes iniciales y proporcionar retroalimentación para el desarrollo futuro.

11.2. Costes temporales

Para contabilizar el tiempo dedicado a este Trabajo de Fin de Grado, se ha estado utilizando la herramienta *Toggl*. A continuación, se puede ver parte de un informe que resume las horas dedicadas al proyecto por meses, desde septiembre hasta julio:

Figura 71. Horas dedicadas al TFG por meses.
(Fuente propia)

En dicho gráfico de barras se aprecia las pocas horas dedicadas en el primer mes del curso, así como los meses de Navidad, donde se trabajó más en el proyecto grupal del último curso de Ingeniería Multimedia. Esto contrasta con los meses finales, en los que se pudo dedicar más horas a este TFG al haber finalizado el resto de asignaturas y trabajos del curso.

En total, tal y como aparece en pequeño en la parte superior izquierda del gráfico, la realización de este Trabajo de Fin de Grado ha requerido la inversión de 460 horas de trabajo.

Gracias a *Toggl*, también se han podido etiquetar las horas contabilizadas según el apartado al que pertenecían y averiguar cuánto tiempo ha necesitado cada sección. En el siguiente gráfico circular, aparece el porcentaje dedicado a cada apartado y también se muestran las horas exactas en la leyenda:

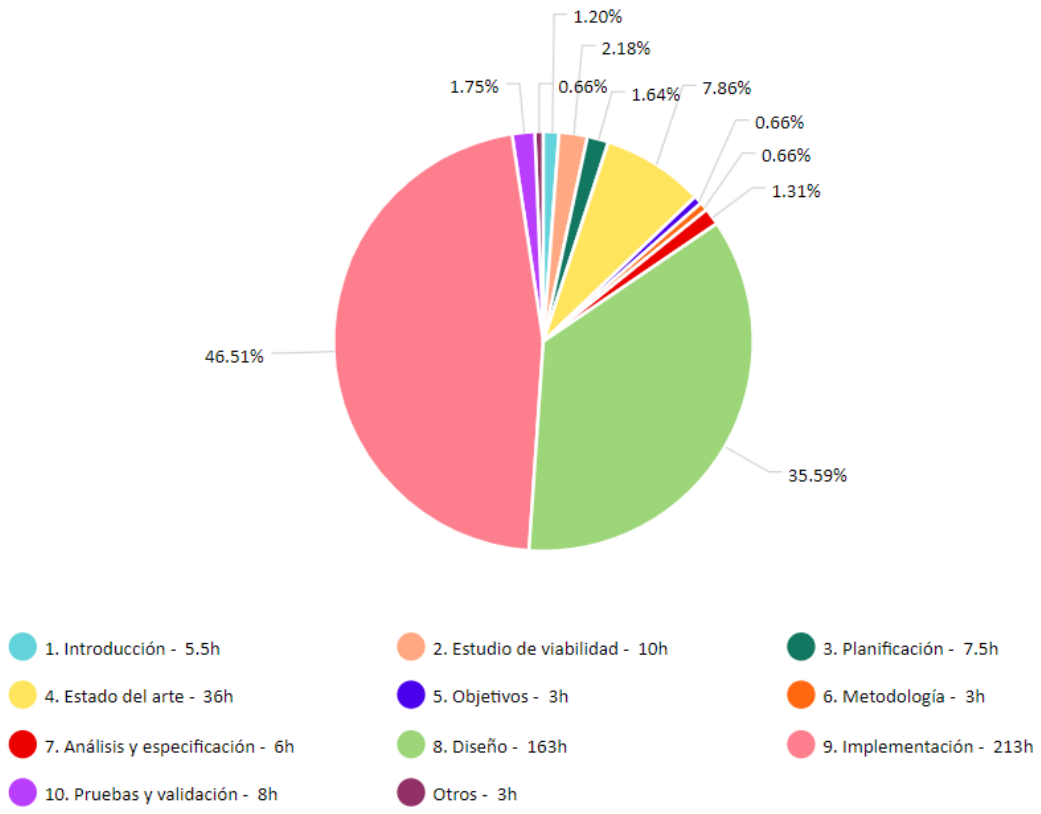


Figura 72. Horas dedicadas al TFG por apartados.
(Fuente propia)

A simple vista se puede distinguir los apartados que más tiempo han necesitado: el del diseño y el de la implementación, con 163 y 213 horas, respectivamente. El 82% de las horas contabilizadas corresponden a estos dos capítulos.

El coste temporal de más de 460 horas supera notablemente las 300 horas que, normalmente, serían necesarias para completar un TFG (ya que equivale a 12 créditos). Esta gran diferencia se debe, sobre todo, a la amplitud del proyecto.

Al definir la idea del TFG a principios de curso, sin experiencia previa desarrollando trabajos de esta extensión, se subestimó el esfuerzo y el tiempo que supondría la realización de este proyecto. Por estos motivos, en los *sprints* finales del desarrollo se tuvo que recortar tanto en interfaces como en funcionalidad.

Si se compara con la planificación inicial, la duración de los primeros apartados del TFG coincidió con las fechas estimadas. Sin embargo, se produjo un retraso de un mes a la hora de comenzar

con la fase de diseño. Esta demora se debe a que, durante las Navidades, se tuvo que dedicar prácticamente todo el tiempo disponible al trabajo grupal.

Además, el apartado de diseño llevó mucho más tiempo del estimado, comenzando en febrero y terminando a mitades de mayo. No obstante, este mes y medio extra se ha podido compensar con el proceso de desarrollo, que finalmente ha durado 2 meses (frente a los 3 estimados).

En resumen, estas más de 460 horas de trabajo han permitido desarrollar buen producto final. Gracias a haber comenzado con el TFG en septiembre y haber trabajado en él de forma constante, este proyecto ha podido llevarse a cabo sin problemas, a pesar del coste temporal.

11.3. Asignaturas relacionadas

A lo largo de todo el grado de Ingeniería Multimedia, se han cursado asignaturas que han sido claves a la hora de realizar este proyecto.

En concreto, durante las primeras etapas de este TFG, los conocimientos adquiridos en las asignaturas de Sistemas de Difusión Multimedia (cuarto curso) y Análisis y Especificación de Sistemas Multimedia (segundo curso) han ayudado a analizar los problemas, a estudiar la viabilidad y a escoger las metodologías a utilizar.

Por otro lado, en cuanto al desarrollo en sí, ha sido de gran utilidad la experiencia adquirida con los proyectos realizados en Programación del Cliente Web (segundo curso) y Desarrollo de Aplicaciones Web (tercer curso).

Por último, han sido significativos los conceptos aprendidos en Usabilidad y Accesibilidad (tercer curso) y Servicios Multimedia Avanzados (cuarto curso), los cuales han permitido diseñar e implementar un producto usable, accesible y que también cuida la experiencia de usuario.

12. Conclusiones y trabajo futuro

En este último capítulo del TFG, se recogen las conclusiones finales, el trabajo a realizar en el futuro y unas pocas impresiones personales.

12.1. Comprobación de objetivos

El objetivo principal para este proyecto fue desarrollar una *app* que permitiese planear y definir los distintos aspectos de un viaje de forma colaborativa, mediante interfaces intuitivas y cuidadas. A falta de terminar las partes pendientes por implementar, se podría afirmar que se ha logrado el propósito de este TFG.

En cuanto a los subobjetivos, se ha cumplido que los datos entre los usuarios estén sincronizados en tiempo real y que la *app* no ocupe mucho espacio (no llega a 10 MB). Este último se puede comprobar, por ejemplo, enviando el APK por *Telegram*:

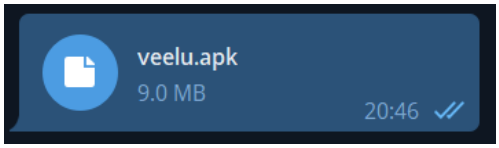


Figura 73. Captura de Telegram donde se muestra el peso final del APK.
(Fuente propia)

Por otra parte, el subobjetivo de la publicación de la *app* en la *Play Store* y el de que los usuarios que la utilicen estén satisfechos se han visto aplazados hasta que se desarrolle del todo la aplicación. Cuando se disponga de una versión completa, probada y segura, se subirá a la *Play Store* para que los amantes de los viajes puedan beneficiarse de ella.

12.2. Trabajo pendiente y posibles mejoras

Como futuro a corto plazo, se tiene pensado continuar con el desarrollo de *veelu*, incluyendo las interfaces y funcionalidad que no ha dado tiempo a implementarse. Una vez esté al completo,

se pasará a probar y pulir la aplicación, así como añadir animaciones y transiciones que hagan más fluida la navegación por la misma.

Además, como trabajo a largo plazo, el hecho de que *veelu* abarque un concepto tan amplio permite que la aplicación pueda expandirse en muchas direcciones. A continuación, se recogen ideas que han surgido durante el desarrollo y recomendaciones de otras personas:

- Añadir las funcionalidades avanzadas pendientes. Este punto hace referencia a herramientas más complejas que no ha dado tiempo a desarrollar, como la previsión meteorológica o los mapas para visualizar la ubicación de las actividades.
- Incluir un chat. Al tratarse de una *app* colaborativa, lo más lógico es que esta permita la comunicación entre sus usuarios por mensajería instantánea. Añadiendo esta funcionalidad, los usuarios serán capaces de prescindir de cualquier otra aplicación para planificar viajes en grupo.
- Orientar la aplicación a una red social vertical. Sería interesante que los usuarios pudiesen compartir una plantilla sobre sus viajes, quitando la información personal que no deseen mostrar al resto. De esta forma, se podría explorar posibles viajes a realizar o también utilizarlos como inspiración.
- Posibilidad de compartir las actividades o fotografías en redes sociales. Continuando con el punto anterior, los usuarios podrían compartir en *Twitter*, *Instagram*... lo que van haciendo durante un viaje, lo cual también ayudaría a promocionar la *app*.
- Exportar la información para crear un álbum o similar. Al finalizar un viaje, se daría la opción de generar una plantilla a partir de los datos del mismo, que podría imprimirse en forma de álbum. Esto aportaría un toque único y original a *veelu*.

En cuanto a la monetización, a pesar de que lo más fácil sería añadir anuncios publicitarios, es mejor apoyarse en un modelo *freemium* para seguir cuidando la experiencia de usuario. Por ejemplo, aquellas personas que pagasen una cuota mensual o anual podrían beneficiarse de las funcionalidades comentadas anteriormente.

De la misma forma, se podrían obtener comisiones mediante acuerdos con agencias de viaje. Estas agencias publicarían viajes planificados que se promocionarían en la *app*. A los usuarios se les recomendaría esas plantillas de viajes, podrían ver las actividades planeadas e información relacionada y, si les interesa, contactar con la agencia para realizar dicho viaje.

12.3. Impresiones personales

La realización de este TFG ha sido un reto al no haber trabajado previamente en un proyecto de semejantes características. A pesar del esfuerzo que ha sido necesario, este trabajo me ha permitido alcanzar subobjetivos más personales como desarrollar una *app* para *smartphones*, trabajar mi faceta de diseñadora y lograr una aplicación de calidad que solventase mi problema de organizar viajes con amigos.

En cuanto a las personas ajenas al proyecto, creo que *veelu* podrá ayudar en gran medida a aquellas que suelen viajar de forma frecuente. En la actualidad, no hay ninguna aplicación que cubra tantas necesidades al planificar un viaje en grupo, por lo que es posible que *veelu* tenga un buen recibimiento entre los jóvenes, mochileros, *influencers* o cualquier persona que prefiera organizar un viaje por su cuenta.

Los apartados que han supuesto una mayor dificultad han sido, como era de esperar, el de diseño y el de desarrollo. En cuanto al primero, he necesitado dedicarle muchas horas a investigar y ponerme al día de las mejores prácticas y tendencias, ya que no tenía experiencia previa. Este proyecto me ha dado la oportunidad de mejorar como diseñadora, tanto de interfaces como de experiencia de usuario.

Sobre el desarrollo, además de habituarme a un *stack* tecnológico nuevo para mí, me ha sido complicado enfrentarme a ciertos problemas, como los relacionados con *Firebase*. No obstante, esto me ha ayudado a adquirir mayor habilidad en cuanto a la resolución de problemas, buscando alternativas y adaptándome a estos contratiempos.

Por otra parte, este TFG también me ha hecho aprender que una buena planificación y el trabajo constante son claves para obtener un producto final de calidad. En definitiva, este Trabajo de Fin de Grado me ha permitido demostrar mis conocimientos aprendidos a lo largo de la carrera y también mejorar mis habilidades como ingeniera.

Referencias

1. Pinto, L. (2013). *Qué es el Dafo y para qué sirve.* Ludiviko Pinto. Disponible en: https://ludivikopinto.com/que-es-es-dafo-y-para-que-sirve/ (Último acceso 13/10/2019)
2. Megias, J. (2012). *Lean canvas, un lienzo de modelos de negocio para startups*. El Blog de Javier Megias. Disponible en: https://javiermegias.com/blog/2012/10/lean-canvas-lienzo-de-modelos-de-negocio-para-startups-emprendedores/ (Último acceso 14/10/2019)
3. Eisenberg, A. (2019). *¿Qué es el marketing de recomendación?* Trusted Shops. Disponible en: https://business.trustedshops.es/blog/marketing-de-recomendacion/ (Último acceso 16/10/2019)
4. Wooman.labs. (2019). *SaveTrip - Travel itinerary & travel expenses*. Google Play. Disponible en: https://play.google.com/store/apps/details?id=com.woomanlabs.mytravelnote (Último acceso 21/10/2019)
5. TripIt Inc. (s.f.). *TripIt: Travel planner*. Google Play. Disponible en: https://play.google.com/store/apps/details?id=com.tripit (Último acceso 21/10/2019)
6. HOTSGO. (2017). *HOTSGO PLAN: Travel planner & travel expenses*. Google Play. Disponible en: https://play.google.com/store/apps/details?id=com.hotsgo.plan (Último acceso 22/10/2019)
7. Lambus GmbH. (2019). *Lambus | Travel planner*. Google Play. Disponible en: https://play.google.com/store/apps/details?id=io.lambus.app (Último acceso 22/10/2019)
8. TravelSpend. (2018). *TravelSpend - Track travel expenses & trip budget*. Google Play. Disponible en: https://play.google.com/store/apps/details?id=tech.jonas.travelbudget (Último acceso 23/10/2019)
9. Wawwo. (2014). *PackPoint: travel packing list.* Google Play. Disponible en: https://play.google.com/store/apps/details?id=com.YRH.PackPoint (Último acceso 24/10/2019)
10. Patro, N. (2018). *Choose the best — Native app vs hybrid app*. Medium. Disponible en: https://codeburst.io/native-app-or-hybrid-app-ca08e460df9 (Último acceso 27/10/2019)
11. Ionic. (s.f.). *Ionic - Cross-Platform Mobile App Development.* Ionic Framework. Disponible en: https://ionicframework.com/ (Último acceso 28/10/2019)
12. Facebook. (s.f.). *React Native · A framework for building native apps using React*. React Native. Disponible en: https://reactnative.dev/ (Último acceso 28/10/2019)

13. Gómez, J. L. (2018). *¿Por qué ha dicho adiós Airbnb a React Native?* Blog Sarenet. Disponible en: https://blog.sarenet.es/airbnb-react-native/ (Último acceso 28/10/2019)
14. Google. (s.f.). *Flutter - Beautiful native apps in record time.* Flutter. Disponible en: https://flutter.dev/ (Último acceso 29/10/2019)
15. Microsoft. (s.f.). *Xamarin | Open-source mobile app platform for .NET*. Microsoft. Disponible en: https://dotnet.microsoft.com/apps/xamarin (Último acceso 29/10/2019)
16. Wang, B. (2016). *The difference between IaaS, PaaS, Baas and SaaS*. Medium. Disponible en: https://medium.com/@benwang_2362/the-difference-between-iaas-paas-baas-and-saas-91133d728917 (Último acceso 06/11/2019)
17. Google. (s.f.). *Firebase - Google*. Firebase. Disponible en: https://firebase.google.com/ (Último acceso 11/11/2019)
18. Mapbox. (s.f.). *Mapbox - An open source mapping platform for custom designed maps*. Mapbox. Disponible en: https://www.mapbox.com/ (Último acceso 16/11/2019)
19. OpenWeather. (s.f.). *Current weather and forecast - OpenWeatherMap*. OpenWeatherMap. Disponible en: https://openweathermap.org/ (Último acceso 16/11/2019)
20. Albares, S. (2019). *Objetivos SMART: Define tus metas sin cometer errores*. BlogsterApp. Disponible en: https://blogsterapp.com/es/objetivos-smart-definir-metas/ (Último acceso 24/11/2019)
21. *Kanban vs. Scrum boards: 11 major differences*. (s.f.). MiroBlog. Disponible en: https://miro.com/blog/scrum-kanban-boards-differences/ (Último acceso 25/11/2019)
22. Atlassian. (s.f.). *Trello*. Trello. Disponible en: https://trello.com/ (Último acceso 26/11/2019)
23. Toggl OÜ. (s.f.). *Toggl - Free Time Tracking Software*. Toggl. Disponible en: https://toggl.com/ (Último acceso 26/11/2019)
24. IEEE. (2008). *Especificación de Requisitos según el estándar de IEEE 830*. Disponible en: https://www.fdi.ucm.es/profesor/gmendez/docs/is0809/ieee830.pdf (Último acceso 04/12/2019)
25. Google. (s.f.). *Firebase & Google Cloud Platform*. Firebase. Disponible en: https://firebase.google.com/firebase-and-gcp (Último acceso 27/02/2020)
26. Loosli, J. (2019). *Node-firestore-import-export*. npm. Disponible en: https://www.npmjs.com/package/node-firestore-import-export (Último acceso 27/02/2020)
27. Qubstudio. (2018). *4 examples of UX personas*. Qubstudio. Disponible en: https://qubstudio.com/blog/4-examples-of-ux-personas/ (Último acceso 28/02/2020)

28. Interactius. (2016). *Metodologías de UX: User Journey map*. Medium. Disponible en: https://blog.interactius.com/metodolog%C3%ADas-de-ux-user-journey-map-c38da9046160 (Último acceso 09/03/2020)
29. Adams, V. (s.f.). *Nunito - Google Fonts*. Google Fonts. Disponible en: https://fonts.google.com/specimen/Nunito (Último acceso 16/03/2020)
30. Pearson, C. (2016). *Typography on the web.* Medium. Disponible en: https://medium.com/rareview/typography-on-the-web-4cd494d6b165 (Último acceso 17/03/2020)
31. Figma. (s.f.). *Figma: the collaborative interface design tool*. Figma. Disponible en: https://www.figma.com/ (Último acceso 06/04/2020)
32. Google. (s.f.). *Firebase products*. Firebase. Disponible en: https://firebase.google.com/products (Último acceso 13/05/2020)
33. Google. (s.f.). *Formularios de Google: crea y analiza encuestas de forma gratuita*. Google. Disponible en: https://www.google.es/intl/es/forms/about/ (Último acceso 14/05/2020)
34. Microsoft. (s.f.). *Visual Studio Code - Code Editing. Redefined*. Visual Studio. Disponible en: https://code.visualstudio.com/ (Último acceso 16/05/2020)
35. Axosoft. (s.f.). *Free Git GUI Client - Windows, Mac, Linux | GitKraken*. GitKraken. Disponible en: https://www.gitkraken.com/ (Último acceso 16/05/2020)
36. Brave Software. (s.f.). *Secure, Fast & Private Web Browser with Adblocker | Brave Browser*. Brave. Disponible en: https://brave.com/ (Último acceso 16/05/2020)
37. Google. (s.f.). *Download Android Studio and SDK tools | Android Developers*. Android Developers. Disponible en: https://developer.android.com/studio (Último acceso 16/05/2020)
38. Ionic. (s.f.). *Capacitor: Cross-platform native runtime for web apps.* Capacitor. Disponible en: https://capacitorjs.com/ (Último acceso 19/05/2020)
39. Wathan, A. (s.f.). *Tailwind CSS - A utility-first CSS framework for rapidly building custom designs*. Tailwind CSS. Disponible en: https://tailwindcss.com/ (Último acceso 19/05/2020)
40. Firebase. (s.f.). *Angular/angularfire: The official Angular library for Firebase*. GitHub. Disponible en: https://github.com/angular/angularfire (Último acceso 20/05/2020)
41. *RxJS - Introduction.* (s.f.). RxJS. Disponible en: https://rxjs-dev.firebaseapp.com/guide/overview (Último acceso 20/05/2020)
42. Bemis, C. (s.f.). *Feather – Simply beautiful open source icons*. Feather Icons. Disponible en: https://feathericons.com/ (Último acceso 25/05/2020)
43. Elasticsearch B.V. (s.f.). *Elasticsearch: The Official Distributed Search & Analytics Engine.* Elastic. Disponible en: https://www.elastic.co/elasticsearch/ (Último acceso 16/06/2020)

44. Algolia. (s.f.). *Algolia Docs.* Algolia. Disponible en: https://www.algolia.com/doc/ (Último acceso 16/06/2020)
45. Ionic. (s.f.). *Push notifications with Firebase in an Ionic/Angular app*. Capacitor. Disponible en: https://capacitorjs.com/docs/guides/push-notifications-firebase (Último acceso 23/06/2020)
46. Fuenzalida, B. (2019). *Entendiendo el plugin push notifications de Capacitor*. Medium. Disponible en: https://medium.com/@brankofuenzalida/entendiendo-el-plugin-push-notifications-de-capacitor-8ca84cdd8d38 (Último acceso 23/06/2020)
47. Vergara, J. (2020). *How to use Firebase callable Cloud Functions with Ionic.* Javebratt. Disponible en: https://javebratt.com/callable-cloud-functions/ (Último acceso 26/06/2020)
48. Google. (2015). *Speed is key: Optimize your mobile experience.* Think with Google. Disponible en: https://www.thinkwithgoogle.com/marketing-resources/experience-design/speed-is-key-optimize-your-mobile-experience/ (Último acceso 08/07/2020)
49. An, D. (2018). *Find out how you stack up to new industry benchmarks for mobile page speed.* Think with Google. Disponible en: https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/ (Último acceso 08/07/2020)
50. Lucas, E. (2019). *How to lazy load in Ionic Angular.* The Ionic Blog. Disponible en: https://ionicframework.com/blog/how-to-lazy-load-in-ionic-angular/ (Último acceso 10/07/2020)
51. Zak, D. (2019). *Ionic capacitor resources generator*. GitHub. Disponible en: https://gist.github.com/dalezak/a6b1de39091f4ace220695d72717ac71 (Último acceso 12/07/2020)